Master's Thesis

Title

Analysis on Evolution of Network-related Functions in the Linux Kernel

Supervisor

Professor Masayuki Murata

Author

Hirotaka Miyakawa

February 6th, 2017

Department of Information Networking

Graduate School of Information Science and Technology

Osaka University

Master's Thesis

Analysis on Evolution of Network-related Functions in the Linux Kernel

Hirotaka Miyakawa

Abstract

With the spread of smartphones and tablet devices, the Internet has become increasingly important in our lives. In recent years, as with IoT (Internet of Things), everything is connected to the Internet and information gathered through things. Services that control and predict devices based on the collected information are under consideration. Recently, NFV (Network Function Virtualization) attracts attention for flexible service development. NFV virtualizes the network function which had been provided by the dedicated devices, and makes it operate as software on general purpose hardware. The virtualization of the network function is to prepare the virtual machine on the hardware and execute the network function implemented as the software on the virtual machine. The network administrator can easily develop a new network function on the network infrastructure, for example, by creating a new virtual machine and executing the network function implemented by the software on the virtual machine. As one application of NFV, edge computing which flexibly expands and arranges functions by NFV in particular, has attracted attention. In edge computing, edge servers are installed near the endhosts at the edge of the network, and (Part of) the processing is performed at the edge server. By using edge computing, the end host only needs to communicate with a relatively close edge server, so that the delay can be reduced and the responsiveness of the service can be enhanced. By performing this edge computing with NFV, it is possible to flexibly expand the functions and scale. However, since an edge server has lower processing performance than a data center, it is not realistic to install all functions on edge servers. Therefore, it is necessary to cut out some processing function (for example, filtering) and place it on the edge. In this thesis, we focused on the Linux kernel implementation, extract commonly used function group (core function), and saw how the core function is used from functions other than core from the viewpoint of graph theory. As a result, Core was used by all components and found to play a fundamental role. Furthermore, by applying the above analysis to multiple versions of the Linux kernel, we analyzed the transition of core functions made by Linux kernel development from the viewpoint of dependency and size between functions, and organize the requirements for extraction and placement of core functions. Core does not change as development progresses, and core is always used when new components are added. On the other hand, it is speculated that ipv4 and irda are important rather than core when network functions are used from the outside of the Linux kernel. Also, the functions in core and ipv4 do not change so much as development progresses.

Keywords

Network Protocol
Protocol Stack
Linux Kernel
Graph Analysis
Software Engineering
Evolusion of Software

Contents

List of Figures

List of Tables

1 Introduction

With the spread of smartphones and tablet devices, the Internet has become increasingly important in our lives. In accordance with diversification of devices and their functions, various services have appeared on the Internet. In recent years, as with IoT (Internet of Things) [1–4], everything is connected to the Internet and information gathered through things. Services that control and predict devices based on the collected information are under consideration. For example, applications such as control of household appliances according to the situation and power transmission control (smart grid) according to electric power demand are studied.

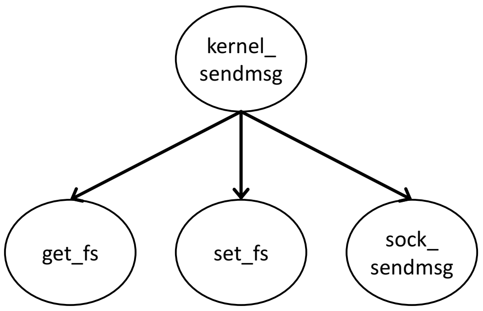
Recently, NFV (Network Function Virtualization) attracts attention for flexible service development [5–8]. NFV virtualizes the network function which had been provided by the dedicated devices, and makes it operate as software on general purpose hardware. The virtualization of the network function is to prepare the virtual machine on the hardware and execute the network function implemented as the software on the virtual machine. The network administrator can easily develop a new network function on the network infrastructure, for example, by creating a new virtual machine and executing the network function implemented by the software on the virtual machine.

As one application of NFV, edge computing [9–12] which flexibly expands and arranges functions by NFV in particular, has attracted attention. In edge computing, edge servers are installed near the endhosts at the edge of the network, and (Part of) the processing is performed at the edge server. As processing, it is conceivable that, for example, collectint traveling data in a specific area and distributint congestion information in ITS. By using edge computing, the end host only needs to communicate with a relatively close edge server, so that the delay can be reduced and the responsiveness of the service can be enhanced. By performing this edge computing with NFV, it is possible to flexibly expand the functions and scale. Currently NFV is being standardized by organizations such as ETSI NFV SGI that develops standard specifications and OPNF to establish reference platform for NFV. Introduction on a commercial basis is also being advanced [13]. However, since an edge server has lower processing performance than a data center, it is not realistic to install all functions on edge servers. Therefore, it is necessary to cut out some processing function (for example, filtering) and place it on the edge.

In this research, we focus on the Linux kernel implementation, extract commonly used function group (functional core), and see how the functional core is used from functions other than core from the viewpoint of graph theory. Furthermore, in this research, by applying the above analysis to multiple versions of the Linux kernel, we analyze the transition of functional core made by Linux kernel development from the viewpoint of dependency and size between functions, and organize the requirements for extraction and placement of functional core. Furthermore, not only static analysis but also analysis is performed in consideration of the frequency of use of network functions. However, since the usage frequency changes according to the supply and demand of network functions, it is difficult to obtain accurate usage frequency. Therefore, statistical values related to protocol utilization are regarded as usage frequency in analysis.

```
int kernel_sendmsg(struct socket *sock,
    struct msghdr *msg,
    struct kvec *vec,
    size_t num, size_t size)
{
    mm_segment_t oldfs = get_fs();
    int result;

    set_fs(KERNEL_DS);
    msg->msg_iov = (struct iovec *)v
    msg->msg_iovlen = num;
    result = sock_sendmsg(sock, msg,
    set_fs(oldfs);
    return result;
}
```



))　　(a) program codes　　(b) a graph representation of function calls

Figure 1: A call graph generated from an example program

2 Graph Representation of the Linux Kernel Implementation

2.1 Call Graph

A call graph is a directed graph with function call as a node, and the relationship in a function call as a link. Figure 1 shows an example of the program code and its call graph. In the program code, the function `kernel_sendmsg` calls the functions `get_fs`, `set_ fs` and `sock_sendmsg`. In the call graph, the four functions are represented as nodes, and edges are drawn from the caller functions to the callee funtions. The calling order in program codes is not reflected in the call graph.

2.2 Obtaining a Call Graph for the Linux Kernel

We use fdump-rtl-expand, one of GCC's debugging options to generate a call graph. CodeViz [14] is one of the tools for generating a call graph from program code and codeviz is provide as a patch to the specific version of the GNU Compiler Collection (GCC). It is commonly used for visualizing program code to understand the structure of. However, even though functions with the same name are declared differently, they are represented as the same node in the call graph generated by Codeviz. GCC consists of a front-end that performs lexical analysis and syntax analysis, and a back-end that generates and optimizes code. The front-end exists for each programming language, but the back end is common to each programming language. Register Transfer Language (RTL) is the intermediate language used when exchanging between the front-end and the back-end in the GCC. The option “fdump-rtl” outputs the RTL in the path specified by the following string. Among them, the option fdump-rtl-expand outputs RTL immediately after its generation when no optimization etc. is performed. The file generated by the option fdump-rtl-expand describes what kind of function is called from what function, and we use it to generate a call graph. The RTL of a simple program (list 1) obtained by the option fdump-rtl-expand is shown in the list 2. The first line of the rtl file is shown below.

```
;; Function foo (foo, funcdef_no=0, [...] symbol_order=0)
```

You can see that the function foo is declared. Similarly, it can be seen from line 55 of the RTL file that the function main is also declared. The 86th line is shown below.

```
(call (mem:QI (symbol_ref:DI ("foo") [...]  ) [0 foo S1 A8])
```

From that line, you can see that main calls foo. Even if the same function is called more than once from a function, it is expressed as an edge of weight 1 on the call graph. This is because we focus on the topology. Some older Linux kernels cannot be compiled with GCC which can output dump files as it is. Therefore, we applied a patch to such Linux kernel, which does not affect function calls so that call graphs can be generated in all versions.

List 1: source code of sample program

```
#include <stdio.h>

int foo(int a)
{
  return a*2;
}

int main (int argc, char *args[])
{
  foo(2);
  return 0;
}
```

List 2: RTL of sample program

```
;; Function foo (foo, funcdef_no=0, decl_uid=1831, cgraph_uid=0, symbol_order=0)

;; Generating RTL for gimple basic block 2

;; Generating RTL for gimple basic block 3

try_optimize_cfg iteration 1

Merging block 3 into block 2...
Merged blocks 2 and 3.
Merged 2 and 3 without moving.
Merging block 4 into block 2...
Merged blocks 2 and 4.
Merged 2 and 4 without moving.
Removing jump 11.
Merging block 5 into block 2...
Merged blocks 2 and 5.
Merged 2 and 5 without moving.

try_optimize_cfg iteration 2

;;
;; Full RTL generated for this function:
;;
(note 1 0 4 NOTE_INSN_DELETED)
(note 4 1 2 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(insn 2 4 3 2 (set (mem/c:SI (plus:DI (reg/f:DI 82 virtual-stack-vars)
                (const_int -4 [0xfffffffffffffffc])) [0 a+0 S4 A32])
        (reg:SI 5 di [ a ])) test.c:1 -1
     (nil))
(note 3 2 6 2 NOTE_INSN_FUNCTION_BEG)
(insn 6 3 7 2 (set (reg:SI 89)
        (mem/c:SI (plus:DI (reg/f:DI 82 virtual-stack-vars)
                (const_int -4 [0xfffffffffffffffc])) [0 a+0 S4 A32])) test.c:2 -1
     (nil))
(insn 7 6 10 2 (parallel [
            (set (reg:SI 87 [ D.1843 ])
                (ashift:SI (reg:SI 89)
                    (const_int 1 [0x1])))
            (clobber (reg:CC 17 flags))
        ]) test.c:2 -1
     (expr_list:REG_EQUAL (ashift:SI (mem/c:SI (plus:DI (reg/f:DI 82 virtual-stack-vars)
                    (const_int -4 [0xfffffffffffffffc])) [0 a+0 S4 A32])
            (const_int 1 [0x1]))
        (nil)))
(insn 10 7 14 2 (set (reg:SI 88 [ <retval> ])
        (reg:SI 87 [ D.1843 ])) test.c:2 -1
     (nil))
(insn 14 10 15 2 (set (reg/i:SI 0 ax)
        (reg:SI 88 [ <retval> ])) test.c:3 -1
     (nil))
(insn 15 14 0 2 (use (reg/i:SI 0 ax)) test.c:3 -1
     (nil))

;; Function main (main, funcdef_no=1, decl_uid=1834, cgraph_uid=1, symbol_order=1)

;; Generating RTL for gimple basic block 2

;; Generating RTL for gimple basic block 3

try_optimize_cfg iteration 1

Merging block 3 into block 2...
Merged blocks 2 and 3.
Merged 2 and 3 without moving.
Merging block 4 into block 2...
Merged blocks 2 and 4.
Merged 2 and 4 without moving.
Removing jump 11.
Merging block 5 into block 2...
Merged blocks 2 and 5.
Merged 2 and 5 without moving.

try_optimize_cfg iteration 2

;;
;; Full RTL generated for this function:
;;
(note 1 0 3 NOTE_INSN_DELETED)
(note 3 1 2 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(note 2 3 5 2 NOTE_INSN_FUNCTION_BEG)
(insn 5 2 6 2 (set (reg:SI 5 di)
        (const_int 2 [0x2])) test.c:6 -1
     (nil))
(call_insn 6 5 7 2 (set (reg:SI 0 ax)
        (call (mem:QI (symbol_ref:DI ("foo") [flags 0x3]  <function_decl 0x7fb13515d1b0
            foo>) [0 foo S1 A8])
            (const_int 0 [0]))) test.c:6 -1
      ...
```

Egypt [15] is a simple perl program to generate a call graph from rtl-dump files. However, it has the same problem as Codeviz. Egypt distinguishes functions only by name. Even though functions with the same name are declared differently, they are represented as the same node in the call graph. It is necessary to use information other than name to distinguish functions which have the same name.

The method to distinguish functions is different between the calling function and the called function. For the calling function, we distinguish the function using the path to the file where the function is declared. Since the rtl-dump file is generated for each source file, we can easily know in which file the function is declared. For the called function, we estimate the callee's function based on following conditions. For example, since function calls from pcibios_fixup_irqs to pcibios_lookup_irq are closed in a single source file arch/x86/pci/irq.c, we can easily estimate the caller funtion. Also, when there is only one calling function, identification is easy. For example, the function calling the function pci_get_device is only pcibios_fixup_irqs. For estimation, we use assembly as well as rtl-dump file. In the Linux kernel, assembly files are used since assembly results in a faster and smaller code [16]. However, the function defined in the assembly file is not included in the rtl-dump file. For example, emulate_vsyscall calls sys_time. Emulate_vsyscall is declared in vsyscall_64.c. On the other hand, Sys_time is not declared in any source file in C language but in the assembly file: syscall64.tbl. Nothed that assembly is architecture dependent and we examine x86-64 in this research.

- The function call is closed in a single file.
- The function is called only at one place in the entire Linux kernel.
- The directories of source files are close where caller function and callee function are declared .
- The function is defined in an assembly.

Even if the estimation is made as described above, function calls that can not specify the caller funtions are still exists. For example, in Linux kernel 3.18.28, there are about 100 function calls that can not specify the caller functions. However, most of these functions are related to file systems which is unrelated to the networking, and we can ignore them in analyzing the calling relationship between network functions.

2.3 Fundamental Property of Linux Kernel Implementation

There are some papers that analyzed the Linux kernel [17–19]. The Linux kernel is analyzed to help develop complex software systems in [17]. The degree distribution revealed that indegree follows power law [20–24] and outdegree follows exponential distribution. In addition, they divide the call graph into modules and analogize the function of each module by the functions contained in each module. They foucus on the ratio of calling / invoking in each function and show that the calling ratio of the basic function is about twice as much as the other functions. The degree distribution of each component is examined in [18], regarding directory as a component. Also, they shows that there is a difference between static components from source code and dynamical components at runtime for connection between components. To know more about how software has changed, some metrics such as degree distribution and clustering coefficient are foucused on, and analysis of the evolution of the Linux kernel is perfomed in [19]. In addition, they propose a method to find out where the change occur due to the property of small-world graph that the average path length is proportional to $\ln N$ (N: number of nodes).

We confirmed whether these properties can be seen also in the latest version. We use Linux kernel 4.7 for analysis. The number of nodes of the call graph is 164,945 and the number of links

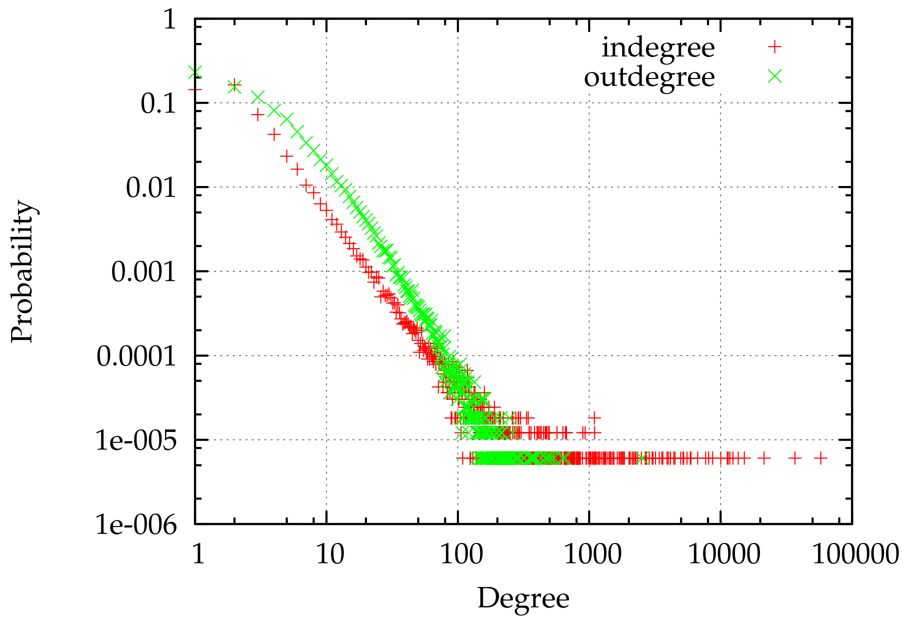


Figure 2: Degree distribution of the call graph for the Linux kernel 4.7

is 946,615. The degree distribution of the call graph generated from the entire Linux kernel is shown in Figure 2. Although the average degree is 8.0, it can be seen some nodes have very large degree. As with the results shown in [17], the distribution of the indegree follows the power law, and the distribution of the outdegree follows the exponential distribution. Even as development of the Linux kernel progresses, the structure of the Linux kernel has not changed significantly. Table 1 shows function names and its degree for top-15 nodes. It is clearly shown that high degree functions are a fundamental function (`printk` for debugging, logging, or CUI-interface) or memory-related functions (`kfree`, `memset`, `memcopy`, and so on).

Figure 3 shows the distribution of path lengths in versions 2.4.0, 2.6.0, 3.0.101, 3.8.13 and 3.16.7. It is increasing from version 2.4 to version 2.6, but it decreases from 2.6 to 3 series. The reason for this is that in version 2.6, there were major changes in IPv6 and IPsec in network implementation. In the early stage of implementation, function calls are not optimized from the viewpoint of speed and readability, and it can be thought that they were rewritten afterwards.

Table 1: High degree functions in the Linux kernel

function's name	degrees
printk	18707
__builtin_expect	17912
kfree	8417
mutex_unlock	5867
spinlock_check	5762
mutex_lock	5331
memset	5318
memcpy	4999
_raw_spin_lock_irqsave	4901
spin_unlock_irqrestore	4723
__builtin_unreachable	4029
__builtin_constant_p	3953
get_current	3771
spin_unlock	3602
spin_lock	3527

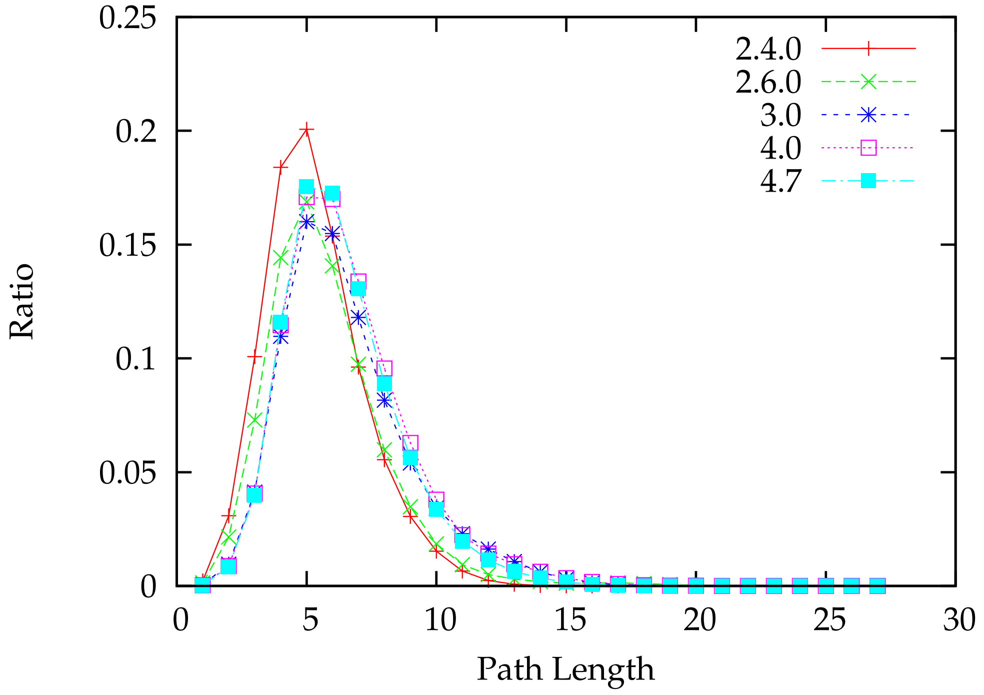


Figure 3: Distribution of path length

3 Characteristics of Network-related Implementation in the Linux Kernel

3.1 Call Graph for Network-related Functions in the Linux Kernel

The Linux kernel supports various functions such as CPU architectures, file systems, and networking. Since our focus is a network-related function, we need to extract the functions related to networking. Fortunately, the directory structure of Linux kernel's files is easy to extract functions related to networking. Files of the Linux kernel are grouped into directories based on their functions. Files related to network functions are gathered in the "net" directory whose sub-directories are also grouped into more specific functions such as "ethernet" and "ipv4". Because the RTL file contains the information on the filepath where the function is declared, we can easily extract a network-related function.

Table 2 shows high degree functions of them. In the table, degrees are much lower than degrees in the Table. 1 and high degree functions are seemed to interfaces or triggers to serve certain network functions.

3.2 Categorizing the Network-related Functions into Protocol Components

In the generated call graph, a function call is a node and a call relation in a function call is a link. However, the function in the program code is too much atomic to analyze the behavior of network functions. We, therefore, categorize functions in the program code into protocol components. For the classification, we use the filepath of the source file. Since the source code of the Linux kernel is divided into directories for each function, we assumed that the functions contained in the same directory are assumed to have the same protocol component here.

3.3 Analysis on Inter-connectivity between Protocol Components

After classifying the function in the program code into several protocol components, we analyze the characteristics of the connection between the protocol components. We consider the directory under the directory "net" as a protocol component and examined the connection between the protocol components. Note that functions declared in the source file directly under the directory "net" shall belong to the component "net".

Table 2: High degree functions related to networking in the Linux kernel

function name	degrees
netdev_open	81
init_one	75
e1000_probe	74
netdev_close	59
ixgbe_probe	56
hci_cmd_complete_evt	56
il4965_pci_probe	53
bond_enslave	52
ieee80211_tx_status	51
igb_probe	50
dev_ethtool	50
rtl_init_one	49
il3945_pci_probe	49
inet6_init	48
ieee80211_do_stop	48

Table 3 shows the number of links between protocol components in Linux kernel 2.4. We use the Linux kernel 2.4 for analysis here since it is easy to analyze as it is simple. The numbers in the table represent the number of calls from the protocol components of the first column to the protocol components of the first row. Many components frequently call themselves and hardly call other protocol components. For example, sunrpc calls its own 192 functions, but functions of other components call only 24 total in total. Protocol components are highly modulared so that an effect of change of functions is less likely to affect other protocol components. In addition, the component "core" is used by all components, and it provides a general-purpose function for each component. For instance, core has sock_cmsg_send, sock_wmalloc and sock_alloc_send.

Noted that the function of TCP is implemented in source files under some directory such as ipv4. In the figure, it is included in the component "ipv4", indicating that TCP and IP are strongly connected.

Table 3: Inter-connectivity between protocol components in the Linux kernel 2.4

	802	appletalk	ax25	bridge	core	ethernet	ipv4	ipx	irda	net	netlink	netrom	packet	rose	sched	sunrpc	unix
802	2	0	0	0	5	0	1	0	0	0	0	0	0	0	0	0	0
appletalk	4	63	0	0	51	0	0	0	0	2	0	0	0	0	0	0	0
ax25	0	0	292	0	61	0	4	0	0	2	0	0	0	0	0	0	0
bridge	0	0	0	118	23	1	0	0	0	0	0	0	0	0	0	0	0
core	0	0	0	0	162	0	1	0	0	4	6	0	0	0	7	0	0
ethernet	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
ipv4	0	0	0	0	382	0	697	0	0	5	7	0	0	0	0	0	0
ipx	6	0	0	0	40	2	0	52	0	2	0	0	0	0	0	0	0
irda	0	0	0	0	207	1	2	0	986	2	0	0	0	0	0	0	0
net	0	0	0	0	20	0	0	0	0	65	0	0	0	0	0	0	0
netlink	0	0	0	0	26	0	0	0	0	2	20	0	0	0	0	0	0
netrom	0	0	22	0	52	0	2	0	0	2	0	129	0	0	0	0	0
packet	0	0	0	0	49	0	0	0	0	0	0	0	5	0	0	0	0
rose	0	0	28	0	58	0	2	0	0	2	0	0	0	157	0	0	0
sched	0	0	0	0	9	0	0	0	0	0	0	0	0	0	4	0	0
sunrpc	0	0	0	0	8	0	0	0	0	16	0	0	0	0	0	192	0
unix	0	0	0	0	37	0	0	0	0	3	0	0	0	0	0	0	38

Table 4 shows the number of links between protocol components in Linux kernel 4.7. Table 5 is an extract of Table 4. The number of components increased from 17 in version 2.4 to 50 in version 4.7. Many components related to wireless and security have been added. In addition, components corresponding to the new communication protocol, such as IPv6, SCTP and MPLS, were also added. Just like version 2.4, many components frequently call themselves and hardly call other protocol components. Many of the newly added protocol components are connected to the protocol components "core" and "ipv4". The number of nodes of core increased from 159 to 1,020, and the number of nodes of ipv 4 increased from 419 to 1,521. Link to core and ipv 4 occupies 77% of links between protocol components. Also, 96% of protocol components use core and 48% of protocol components use ipv4. Protocol components core and ipv4 became the functional core. We analyze the evolutionary process in the next section.

Table 4: Inter-connectivity between protocol components in the Linux kernel 4.7

	802	appletalk	ax25	bridge	core	ethernet	ipv4	ipx	irda	net	netlink	netrom	packet	rose	sched	sunrpc	unix	8021q	bluetooth	key	llc	xfrm	9p	atm
802	53	0	0	0	42	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9	0	0	0
appletalk	4	61	0	0	72	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ax25	0	0	268	0	120	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
bridge	3	0	0	604	233	1	15	0	0	4	5	0	0	0	3	0	0	5	0	0	1	0	0	0
core	0	0	0	0	1516	5	6	0	0	18	49	0	0	0	19	0	0	1	0	0	0	7	0	0
ethernet	0	0	0	0	9	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ipv4	0	0	0	0	1130	2	2014	0	0	46	25	0	0	0	0	0	0	0	0	0	0	72	0	0
ipx	6	0	0	0	56	0	0	65	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
irda	0	0	0	0	338	2	0	0	939	4	4	0	0	0	0	0	0	0	0	0	0	0	0	0
net	0	0	0	0	28	0	0	0	0	102	0	0	0	0	0	0	0	0	0	0	0	0	0	0
netlink	0	0	0	0	78	0	0	0	0	4	130	0	0	0	0	0	0	0	0	0	0	0	0	0
netrom	0	0	22	0	99	0	0	0	0	4	0	117	0	0	0	0	0	0	0	0	0	0	0	0
packet	0	0	0	0	117	0	1	0	0	6	3	0	56	0	0	0	0	0	0	0	0	0	0	0
rose	0	0	28	0	82	0	0	0	0	4	0	0	0	138	0	0	0	0	0	0	0	0	0	0
sched	0	0	0	0	427	1	1	0	0	1	11	0	0	0	927	0	0	1	0	0	0	0	0	0
sunrpc	0	0	0	0	41	0	1	0	0	39	0	0	0	0	0	1281	0	0	0	0	0	0	0	0
unix	0	0	0	0	79	0	0	0	0	8	3	0	0	0	0	0	82	0	0	0	0	0	0	0
8021q	12	0	0	0	77	1	0	0	0	2	0	0	0	0	4	0	0	54	0	0	0	0	0	0
bluetooth	0	0	0	0	406	2	0	0	0	26	0	0	0	0	0	0	0	0	2136	0	0	0	0	0
key	0	0	0	0	48	0	0	0	0	4	0	0	0	0	0	0	0	0	0	85	0	66	0	0
llc	0	0	0	0	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0
xfrm	0	0	0	0	71	0	1	0	0	2	36	0	0	0	0	0	0	0	0	0	0	324	0	0
9p	0	0	0	0	2	0	0	0	0	7	0	0	0	0	0	0	0	0	0	0	0	0	179	0
atm	0	0	0	0	156	5	4	0	0	8	0	0	0	0	4	0	0	0	0	0	0	0	0	148
batman-adv	0	0	0	0	142	4	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
can	0	0	0	0	85	0	0	0	0	5	3	0	0	0	0	0	0	0	0	0	0	0	0	0
dcb	0	0	0	0	15	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
dns_resolver	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ipv6	0	0	0	0	989	1	165	0	0	40	13	0	0	0	0	0	0	0	0	0	0	91	0	0
l2tp	0	0	0	0	130	1	17	0	0	13	16	0	0	0	0	0	0	0	0	0	0	3	0	0
mac80211	0	0	0	0	344	5	3	0	0	0	0	0	0	0	13	0	0	0	0	0	0	0	0	0
netfilter	0	0	0	1	431	0	49	0	0	30	74	0	0	0	0	0	0	0	0	0	0	3	0	0
netlabel	0	0	0	0	17	0	30	0	0	0	15	0	0	0	0	0	0	0	0	0	0	0	0	0
rfkill	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
wireless	0	0	0	0	311	0	1	0	0	4	108	0	0	0	0	0	0	0	0	0	0	0	0	0
6lowpan	0	0	0	0	26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ceph	0	0	0	0	3	0	0	0	0	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0
dccp	0	0	0	0	167	0	63	0	0	9	0	0	0	0	0	0	0	0	0	0	0	2	0	0
dsa	0	0	0	0	65	3	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0
ieee802154	0	0	0	0	171	0	9	0	0	9	21	0	0	0	0	0	0	0	0	0	0	0	0	0
mac802154	0	0	0	0	55	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
mpls	0	0	0	0	48	0	1	0	0	4	1	0	0	0	0	0	0	0	0	0	0	0	0	0
nfc	0	0	0	0	358	0	0	0	0	4	42	0	0	0	0	0	0	0	0	0	0	0	0	0
openvswitch	0	0	0	0	157	2	4	0	0	0	16	0	0	0	0	0	0	0	0	0	0	1	0	0
rds	0	0	0	0	44	0	3	0	0	21	0	0	0	0	0	0	0	0	0	0	0	0	0	0
sctp	0	0	0	0	154	0	20	0	0	11	2	0	0	0	0	0	0	0	0	0	0	1	0	0
switchdev	0	0	0	0	28	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
tipc	0	0	0	0	201	0	7	0	0	17	19	0	0	0	0	0	0	0	0	0	0	0	0	0
vmw_vsock	0	0	0	0	58	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
kcm	0	0	0	0	63	0	1	0	0	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	batman-adv	can	dcb	dns_resolver	ipv6	l2tp	mac80211	netfilter	netlabel	rfkill	wireless	6lowpan	ceph	dccp	dsa	ieee802154	mac802154	mpls	nfc	openvswitch	rds	sctp	switchdev	tipc	vmw_vsock	kcm
802	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
appletalk	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ax25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
bridge	0	0	0	0	11	0	0	115	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15	0	0	0
core	0	0	0	0	1	0	0	3	0	0	3	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0
ethernet	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ipv4	0	2	0	0	10	0	0	335	6	0	0	0	0	0	0	0	0	0	0	0	0	0	6	0	0	0
ipx	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
irda	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
net	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
netlink	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
netrom	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
packet	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
rose	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
sched	0	0	0	0	0	0	0	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
sunrpc	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
unix	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8021q	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
bluetooth	0	0	0	0	0	0	0	0	0	6	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
key	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
llc	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
xfrm	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9p	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
atm	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
batman-adv	923	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
can	0	78	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
dcb	0	0	33	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
dns_resolver	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ipv6	0	0	0	0	1583	0	0	228	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
l2tp	0	0	0	0	28	93	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
mac80211	0	0	0	0	2	0	1763	0	0	0	193	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
netfilter	0	0	0	0	44	0	0	2563	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
netlabel	0	0	0	0	0	0	0	0	114	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
rfkill	0	0	0	0	0	0	0	0	0	43	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
wireless	0	0	0	0	0	0	0	0	0	13	661	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6lowpan	0	0	0	0	1	0	0	0	0	0	0	42	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ceph	0	0	0	0	0	0	0	0	0	0	0	0	668	0	0	0	0	0	0	0	0	0	0	0	0	0
dccp	0	0	0	0	26	0	0	1	0	0	0	0	0	324	0	0	0	0	0	0	0	0	0	0	0	0
dsa	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8	0	0	0	0	0	0	0	0	0	0	0
ieee802154	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	97	0	0	0	0	0	0	0	0	0	0
mac802154	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	14	74	0	0	0	0	0	0	0	0	0
mpls	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	35	0	0	0	0	0	0	0	0
nfc	0	0	0	0	0	0	0	0	0	6	0	0	0	0	0	0	0	0	575	0	0	0	0	0	0	0
openvswitch	0	0	0	0	6	0	0	24	0	0	0	0	0	0	0	0	0	0	0	374	0	0	0	0	0	0
rds	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	409	0	0	0	0	0
sctp	0	0	0	0	27	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1068	0	0	0	0
switchdev	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	49	0	0	0
tipc	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	544	0	0
vmw_vsock	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	120	0
kcm	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	50

Table 5: Inter-connectivity between protocol components in the Linux kernel 4.7 (selected)

	bridge	core	ipv4	irda	sched	sunrpc	bluetooth	ipv6	netfilter	wireless	mac80211	sctp
bridge	604	233	15	0	3	0	0	11	115	0	0	0
core	0	1516	6	0	19	0	0	1	3	3	0	0
ipv4	0	1130	2014	0	0	0	0	10	335	0	0	0
irda	0	338	0	939	0	0	0	0	0	0	0	0
sched	0	427	1	0	927	0	0	0	9	0	0	0
sunrpc	0	41	1	0	0	1281	0	3	0	0	0	0
bluetooth	0	406	0	0	0	0	2136	0	0	0	0	0
ipv6	0	989	165	0	0	0	0	1583	228	0	0	0
netfilter	1	431	49	0	0	0	0	44	2563	0	0	0
wireless	0	311	1	0	0	0	0	0	0	661	0	0
mac80211	0	344	3	0	13	0	0	2	0	193	1763	0
sctp	0	154	20	0	0	0	0	27	1	0	0	1068

3.4 Functional Core in the Linux Kernel

In previous analysis, we investigate which protocol component plays a central role to provide network functions in network-related functions. Then, we examine the protocol components that are important when external functions use network functions. Table 6 shows the number of links from out of network functions to the protocol components providing network functions. Note that we exclude some protocol components which are considered to be network-related under the “drivers” directory. That’s because some functions under the “drivers” directory provide network functions but now we focus on functions that use network functions. In the table, sunrpc responsible for interprocess communication is called from outside frequently, but other components including core are rarely called. The network function seems not to be used much inside the Linux kernel because it is used from external applications.

However, we can not examine function calls from outside the Linux kernel for all applications. Therefore, we investigate which protocol components are used frequently when network functions are used from the out of the Linux kernel, paying attention to functions with indegree 0. The fact that the indegree is 0 means that it is not called from any function in the Linux kernel, and such a function is thought to be an interface with the outside of the Linux kernel. Table 7 shows the number of nodes whose indegree is 0 for each protocol components. There are many nodes with indegree 0 in irda and ipv4. When using the network function, it is considered that other components is used as an interface instead of core frequently called in connections between protocol components and core behaves as functional core.

Table 6: Number of links from network-unrelated functions

protocol components	number of links from out of network functions
net	4
802	0
appletalk	0
ax25	0
bridge	0
core	3
ethernet	0
ipv4	0
ipx	0
irda	0
netlink	0
netrom	0
packet	0
rose	0
sched	0
sunrpc	175
unix	0

Table 7: Number of nodes whose indegree is 0

protocol components	number of nodes with indegree 0
net	8
802	6
appletalk	28
ax25	31
bridge	10
core	36
ethernet	2
ipv4	167
ipx	25
irda	258
netlink	7
netrom	35
packet	13
rose	40
sched	5
sunrpc	53
unix	20

4 Evolution of Network-related Implementation in the Linux kernel

We investigate the changes of interconnectivity related to network functions in the development of the Linux kernel. Version 2.4.0 (January 2001) to version 4.7 (July 2016) are subject to investigation. Notable changes during the development are supports of IPv6 (v3.0), VPN (v3.0), and IPsec (v2.6). Needless to say, other network functions are also developed intensively; for example, mobile ad-hoc networking (B.A.T.M.A.N) and Stream Control Transmission Protocol (SCTP). A full of ChangeLog is available at [25].

4.1 Changes of Topological Characteristics

As the development of the Linux kernel progresses, we examine how the topology properties of the call graph are changing.

Figure 4 shows the change in the number of nodes and the number of links in the call graph consisting of functions related to the network. The horizontal axis is the number of days elapsed since the release date of the oldest version to be analyzed. Since new function calls have been added without deleting old function calls, the number of nodes and the number of links are increasing as the development progresses. Due to the support of IPsec and IPv6, the number of nodes and the number of links are rapidly increasing from version 2.4.0 to 3. IPsec and IPv6 have been implemented since version 2.4.0, but it is experimental. In this research, we deal only with results based on the default configuration. In version 3 and later, the change was large from 3.7.10 to 3.9.11, the number of nodes increased from 18,758 to 24,240, and the number of links increased from 74,869 to 98,279. This is because Stream Control Transmission Protocol (SCTP) [26] is enabled by default from version 3.8.0. An increase in the number of links is remarkable compared to the number of nodes, which means that functions are being actively reused. In general, reuse in software is important for promoting development, but this leads to an increase in reliance among functions, which makes it difficult to divide functions.

Figure 5 shows the degree distribution in version 2.4.0, 2.6.0, 3.0.101, 3.8.13, 3.16.7. Here, we considered that both caller and callee were related to functions, and calculated the order distribution with undirected graph. From Figure 5, all versions follow power raw. That is, while most functions are called from several functions, the number of calls to fewer functions is significantly greater than the average. Functions with higher degrees continue to be high degrees, suggesting

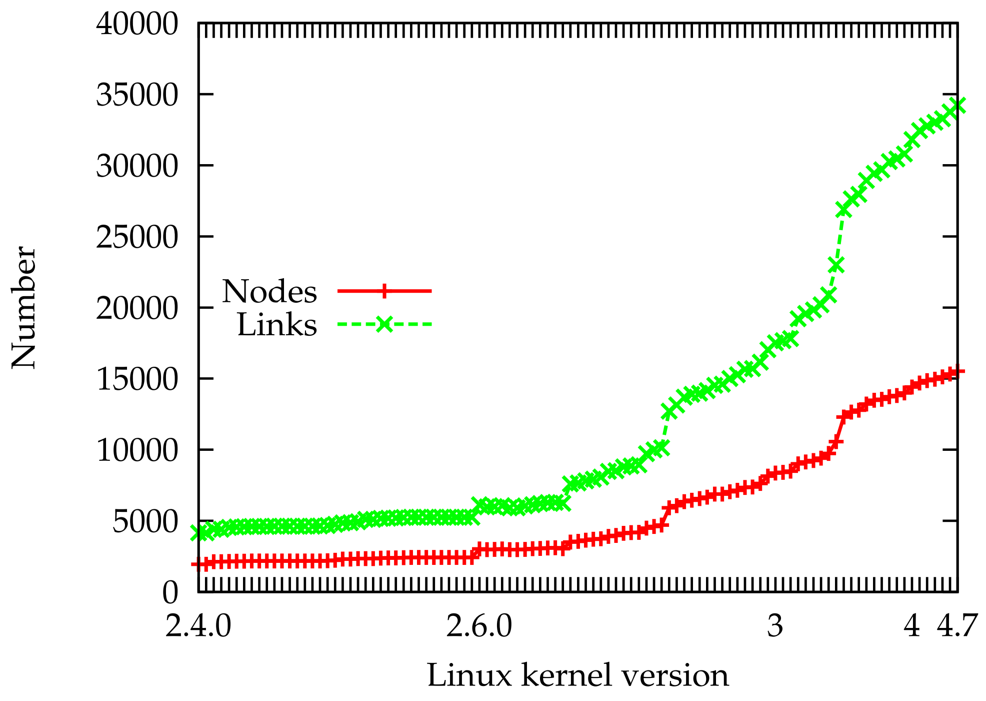


Figure 4: Changes of the number of nodes and links for each kernel

that new functions tend to use functions with large orders.

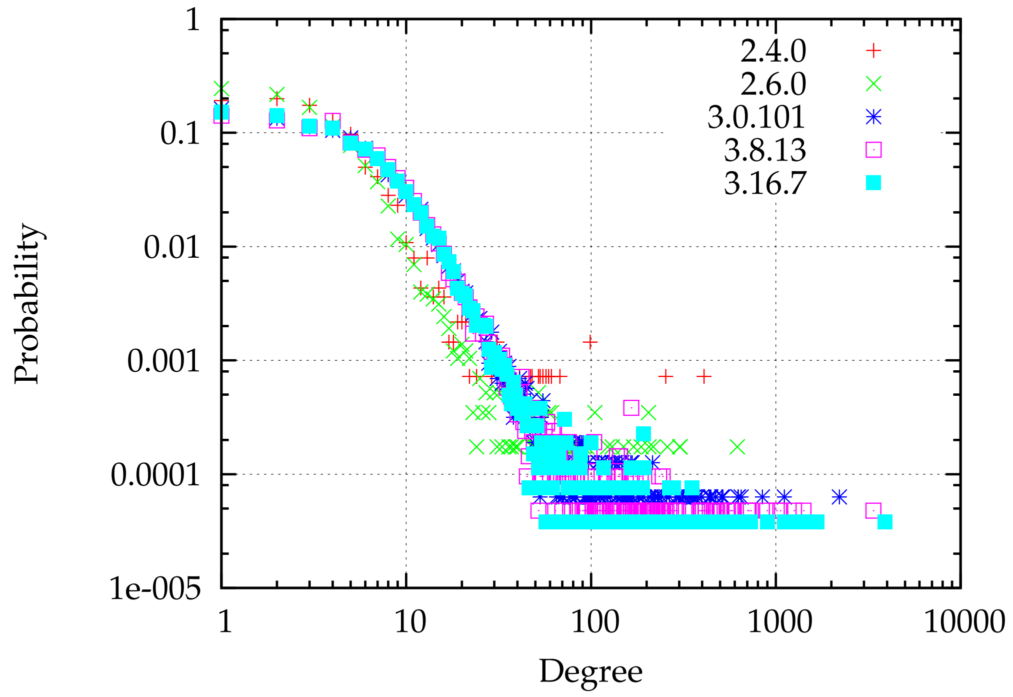


Figure 5: Changes of degree distribution for each kernel

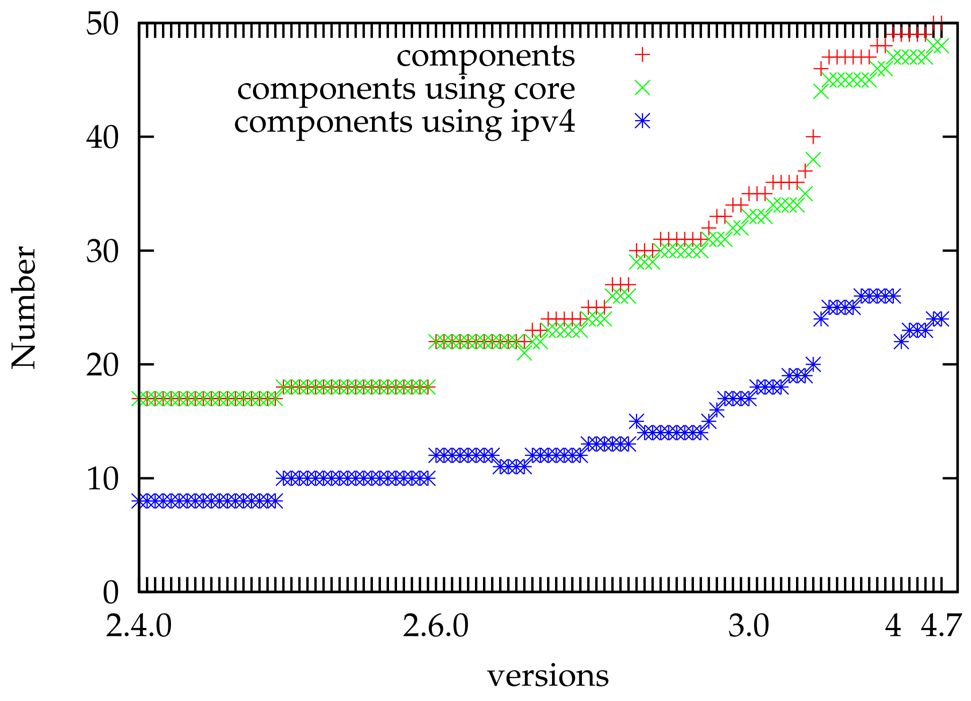


Figure 6: Changes of numbers of protocol components

4.2 Changes of the Inter-connectivity between Protocol Components

Figure 7 shows the connections between protocol components in the Linux kernels 3.0 and 4.7. The size of the node expresses the number of calls in the component, and the thickness of the edge expresses the number of calls between the protocol components. The number of components increased from 35 to 50 by about 1.5 times, and the number of links between components also increased. It seems that these correspond to addition of functions by development progress. All of the newly added protocol components are connected to the protocol component "core" and 60% of them are connected to the protocol component "ipv4". Many components depend on these components and are frequently used when new features are added. Figure 6 shows the transition of the number of total protocol components and the number of protocol components using ipv4 and core. Components that use core and ipv4 are increasing as the total number of components increases. Core and ipv4 are almost always used from new components when networking functionality is added. Thus, the components "core" and "ipv4" play a key role in providing network functions and are considered as functional core.

Also, we investigate the changes of interdependency of network functions using the modu-

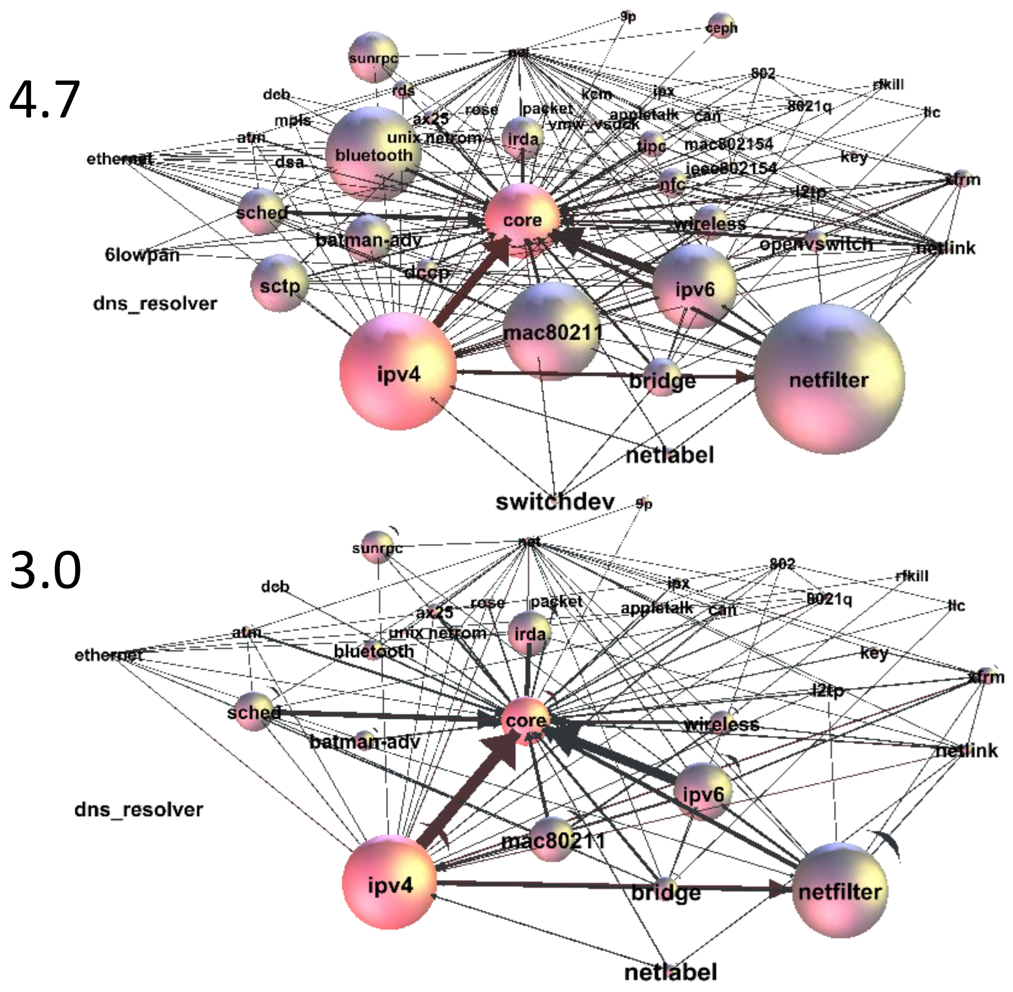


Figure 7: Connection between protocol components in the Linux kernel 4.7 and 3.0

larity metric [27] to understand the interdependency of protocol components. The modularity is a measure of the quality that represents the gap between the number of inner-module links and the number of inter-module links. The modularity takes 0.5 when the ratio of the number of inter-module links and the number of inner-module links is (statistically) equals to the randomly generated graph. The modularity increases as the ratio increases and vice versa.

We divide a call graph into modules, which consists of a set of nodes, to calculate the modularity metric. We consider two scenarios for division; graph-theoretical division and function-based division. The graph-theoretical division is obtained by applying Louvain method [28] which divides a graph into modules such that the modularity metric is highest. The modularity metric based on the Louvain method captures whether the modular software design, which is the basic principles in software development, is performed or not. In general, high modular software is easy to separate the functionality of a software into interdependent, interchangeable modules [29]. The function-based division is our own approach to dividing the call graph into groups based on the protocol component that a function belongs to. Then, we calculate the modularity metric by regarding a set of functions belonging to a protocol component as a module. The function-based division captures rather a semantic relation of a call graph; whether network functions are easy to separate into interdependent, interchangeable functions or not.

Figure 8 shows the changes of modularity dependent on the kernel versions. In the figure, the modularity metric calculated by the function-based division is always lower than the modularity metric calculated by the graph-theoretical division. This indicates that interdependency between the network functions is high, and therefore a lot of effort may require separating the network function for the network function virtualization. The kernel development itself will benefit from modular software design thanks to the high modularity value. However, the interdependency or interchangeability of network functions goes toward a bad direction as the difference of the modularity metrics is getting larger as the development of the Linux kernel progress. The more sophisticated design concept for both software engineering and networking perspective may be necessary to promote the upcoming network function virtualization.

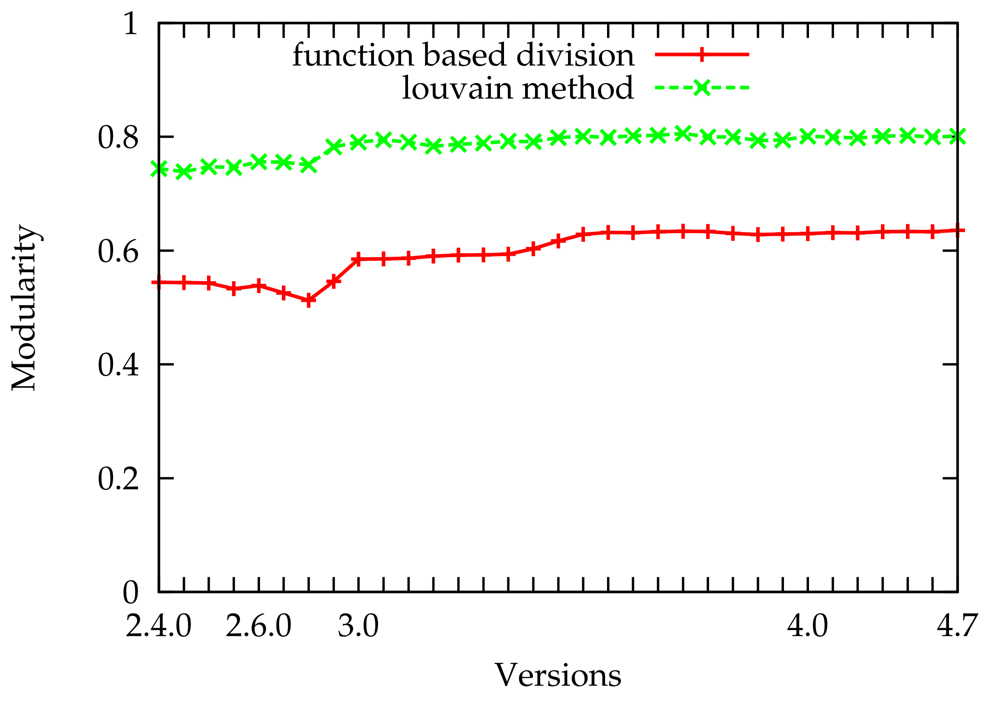


Figure 8: Changes of modularity

4.3 Evolution of Functional Core

Figure 9 shows the changes in the number of nodes in the net and component sizes in core, ipv4, ipv6, mac80211 and netfilter. Until around 2.6.24, the size in ipv4 and core increased as the total number of nodes increased while the increasing speed of size in ipv4 and core decreased after 2.6.24. In place of ipv4 and core, the size in newly added components such as ipv6 increases as the total number of nodes increases. Regardless of the increase in the total number of nodes, the size in irda has not changed very much. In deploying newer components to an edge server, it is necessary to prepare for future increase in size. On the other hand, in deploying older protocol components such as ipv4 and core, we don't need much margin even if they are commonly used.

We will investigate what extent protocol components changes as development progresses and new functions are added. Changes of the number of unchanged functions in each component for each version are shown in the Table 8. Here, a function that does not change means that the calling function is the same as the previous version. If the function does not exist in the previous version, it is not counted as an unchanged function. Figure 10 is an extract of the Table 8. The unchanged function in core, ipv4, ipv6, sched,sunrpc, bluettoh, mac80211 and netfilter has increased greatly. For example, the unchanged function in core has increased from 159 in 2.4 to 979 in 4.7. The number of unchanged functions increases as component size increases. Many functions continue to be used without being changed. In adding some functions to deployed component, a few changes to existing parts of the component are required.

Next, we examined how the core responds to component additions. When a new component was added, I checked whether the function of the core that the added component calls exists in the previous version. 99% of the functions of core called from the new component already existed in the previous version. These functions occupy 15% of the whole core (in version 4.7), and it is possible that only some of the functions in the core are functional core. The number of nodes called in new component added and rest of core are shown in Figure 11. The number of nodes called from the new component does not increase even as the development progresses. Only a part of the core is the functional core and it unchanges as the development progresses.

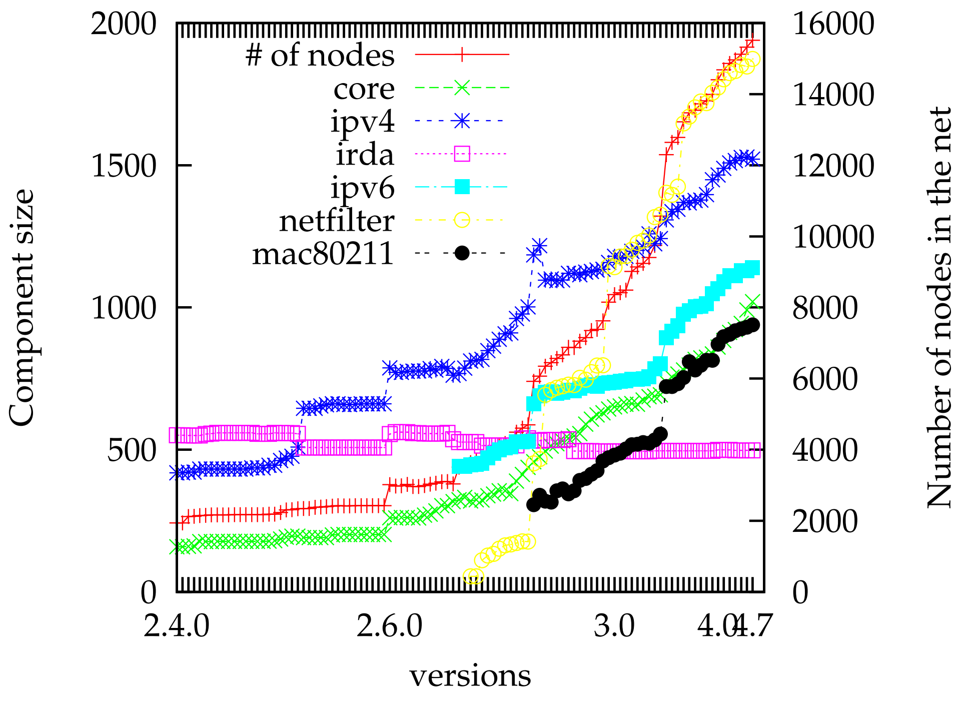


Figure 9: Changes of the number of the nodes and component size

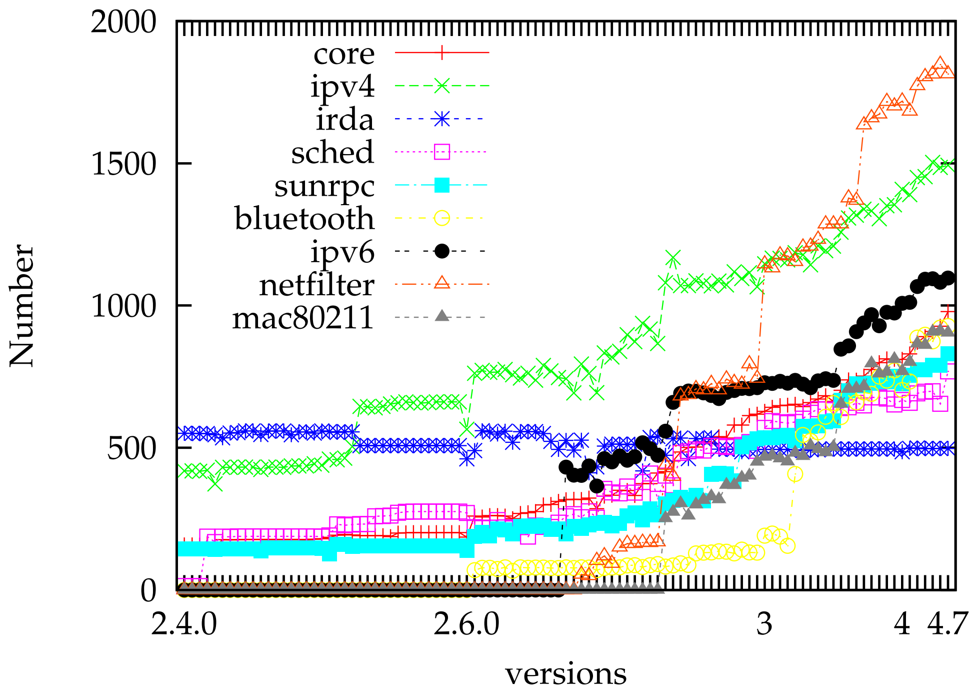


Figure 10: Changes of the number of unchange functions

Table 8: unchanged functions in each protocol component

version	number of nodes	number of unchanged nodes	802	appletalk	ax25	bridge	core	ethernet	ipv4	ipx	irda	net	netlink	netrom	packet	rose	sched	sunrpc	unix	8021q	bluetooth	key	llc	xfrm	ipv6	netfilter	netlabel	rfkill	wireless	atm	can	mac80211	dcb	l2tp	dns_resolver	batman-adv	9p	openvswitch	ceph	dsa	nfc	sctp	dccp	ieee802154	mac802154	rds	tipc	vmw_vsock	mpls	6lowpan	switchdev	kcm
2.4.0	1944	1944	13	63	150	61	159	5	419	49	551	0	19	97	13	114	14	145	31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2.6.0	3018	2154	11	42	112	57	186	5	565	20	461	0	22	63	18	92	271	140	27	23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2.6.10	3103	3049	15	63	132	186	299	6	789	55	550	0	17	80	12	94	222	228	5	21	79	37	11	92	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2.6.20	4175	3928	16	55	118	203	351	8	839	46	513	0	57	68	0	85	340	235	5	26	80	35	8	113	471	150	42	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2.6.30	6567	6184	36	60	130	218	501	11	1070	49	462	0	78	81	18	88	488	318	49	47	88	54	9	201	700	686	91	24	94	123	63	261	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2.6.31	6666	6444	36	60	130	216	503	12	1087	49	533	0	61	81	18	89	494	333	49	48	129	54	9	201	699	707	92	24	141	137	65	300	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3.0	8361	8038	36	55	132	278	628	10	1147	44	494	0	71	81	28	89	570	535	53	44	191	53	9	215	729	1146	95	44	331	143	61	468	17	19	3	130	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3.1	8417	8240	36	55	132	288	642	10	1167	44	494	0	71	81	22	89	594	536	53	46	199	53	8	215	726	1132	95	44	357	143	61	472	15	20	3	156	85	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3.2	8488	8323	36	55	132	297	648	10	1166	44	494	0	71	81	28	89	590	541	51	43	188	53	8	214	734	1177	94	44	361	143	64	462	25	19	3	179	82	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3.3	9012	8275	36	55	132	287	650	10	1161	44	494	0	64	81	33	89	587	544	53	41	155	53	8	216	727	1174	95	44	358	140	73	451	25	20	3	188	91	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3.4	9137	8676	36	55	132	301	653	10	1186	44	493	0	68	81	37	89	591	524	60	50	408	53	8	206	737	1154	87	44	345	143	72	482	18	20	3	199	90	99	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3.5	9241	8912	36	53	130	295	645	10	1178	42	490	0	74	79	37	87	602	576	59	49	545	53	8	215	724	1205	95	44	381	144	73	470	24	18	3	176	91	104	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3.6	9405	8971	36	55	130	303	659	10	1143	42	494	0	76	81	37	89	623	575	59	48	524	53	8	216	711	1208	95	44	368	142	72	507	19	21	3	226	93	104	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3.7	9742	9163	36	53	132	305	671	7	1210	46	493	0	73	81	36	89	636	577	60	47	554	53	9	215	735	1232	95	44	397	144	75	494	37	21	3	221	92	93	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3.8	10571	9485	36	56	132	299	683	7	1193	48	496	0	76	81	42	89	635	591	61	51	609	53	10	215	743	1286	95	44	385	142	75	484	37	20	3	203	80	106	222	0	0	0	0	0	0	0	0	0	0	0	0	0
3.9	12297	10212	36	56	132	305	663	6	1209	46	496	0	76	81	40	89	632	595	61	43	657	52	10	194	737	1286	95	44	395	146	75	502	37	19	3	254	80	108	207	16	181	457	0	0	0	0	0	0	0	0	0	0
3.11	12643	11698	55	56	132	330	702	9	1258	49	495	0	63	81	36	89	640	667	61	57	608	56	10	232	847	1286	92	46	375	146	75	654	37	101	3	237	99	83	209	16	167	459	225	74	21	326	253	92	0	0	0	0
3.12	12784	12418	54	56	132	353	739	9	1308	49	496	0	90	81	44	89	642	700	59	57	683	58	10	233	858	1378	95	46	415	146	75	706	37	102	3	284	102	108	258	16	276	446	223	73	22	326	279	96	3	0	0	0
3.13	13219	12480	57	56	132	352	738	9	1318	49	495	0	88	81	45	89	658	725	61	56	660	58	10	239	908	1369	92	46	414	146	75	708	37	102	3	247	102	130	258	16	269	465	226	75	22	325	273	96	3	0	0	0
3.14	13476	12896	57	56	132	349	752	8	1339	49	496	0	82	81	40	89	649	719	61	56	695	57	10	237	939	1634	95	46	408	144	74	715	36	99	3	301	102	126	250	16	332	466	226	71	22	326	252	96	2	0	0	0
3.15	13544	13201	57	56	132	359	775	9	1333	49	496	0	83	81	46	89	677	726	61	55	690	58	10	234	968	1659	95	46	488	146	75	796	36	101	3	329	102	137	259	16	331	463	225	65	19	326	242	95	2	0	0	0
3.16	13728	13153	57	56	132	345	799	9	1305	50	496	0	87	81	46	89	677	733	61	57	750	58	10	237	929	1673	95	46	437	145	77	760	37	94	3	353	102	122	268	16	336	457	216	92	21	326	212	96	3	0	0	0
3.17	13813	13493	57	56	132	386	813	9	1351	50	496	0	85	81	48	89	673	732	61	58	736	58	10	245	976	1717	95	46	462	145	78	766	37	99	3	361	102	135	241	16	339	467	224	125	71	326	237	96	3	0	0	0
3.18	13995	13559	57	56	132	377	811	9	1354	49	495	0	87	81	46	89	652	752	61	57	772	58	10	245	974	1701	98	46	461	137	77	811	37	98	3	361	102	135	268	14	346	465	224	122	70	324	229	96	3	4	0	0
4	14407	13445	57	55	131	393	811	10	1408	46	486	0	81	80	46	88	668	725	57	57	703	57	10	248	1008	1718	95	46	443	145	78	769	37	90	3	360	102	124	271	25	327	463	223	116	70	325	190	95	3	4	0	0
4.2	14687	13742	52	55	129	402	830	13	1390	50	498	0	79	81	44	88	658	755	60	53	733	58	9	245	1011	1683	98	46	482	146	77	801	37	100	3	358	100	191	277	16	381	464	213	125	84	319	238	96	3	2	8	0
4.3	14860	14420	58	56	132	418	871	12	1450	51	499	0	90	80	48	88	691	774	66	58	887	58	10	248	1067	1772	98	46	495	146	76	866	37	100	3	354	98	164	281	32	389	464	222	132	87	309	270	95	19	24	27	0
4.4	14963	14515	53	55	125	410	891	12	1453	50	497	0	90	79	46	87	697	774	59	57	897	58	9	246	1093	1804	98	46	500	146	75	861	37	101	3	357	98	184	277	27	384	464	214	129	93	325	278	98	26	25	25	0
4.5	15122	14773	56	56	132	447	909	12	1503	51	498	0	91	80	52	88	699	790	71	58	874	58	10	245	1094	1813	98	46	496	146	76	907	36	102	3	348	97	199	282	30	399	460	223	137	95	334	277	98	31	25	40	0
4.6	15325	14711	56	54	132	436	927	14	1486	51	498	0	91	80	48	88	655	789	72	58	922	58	10	245	1082	1847	98	45	493	146	76	911	36	102	3	224	97	189	278	33	401	477	223	139	95	323	306	98	33	41	43	0
4.7	15516	15071	56	55	133	446	979	14	1493	51	498	0	87	80	50	87	769	830	71	56	928	58	10	245	1097	1814	98	43	508	146	76	904	36	100	3	384	99	202	265	31	401	480	220	136	94	272	298	97	35	48	43	44

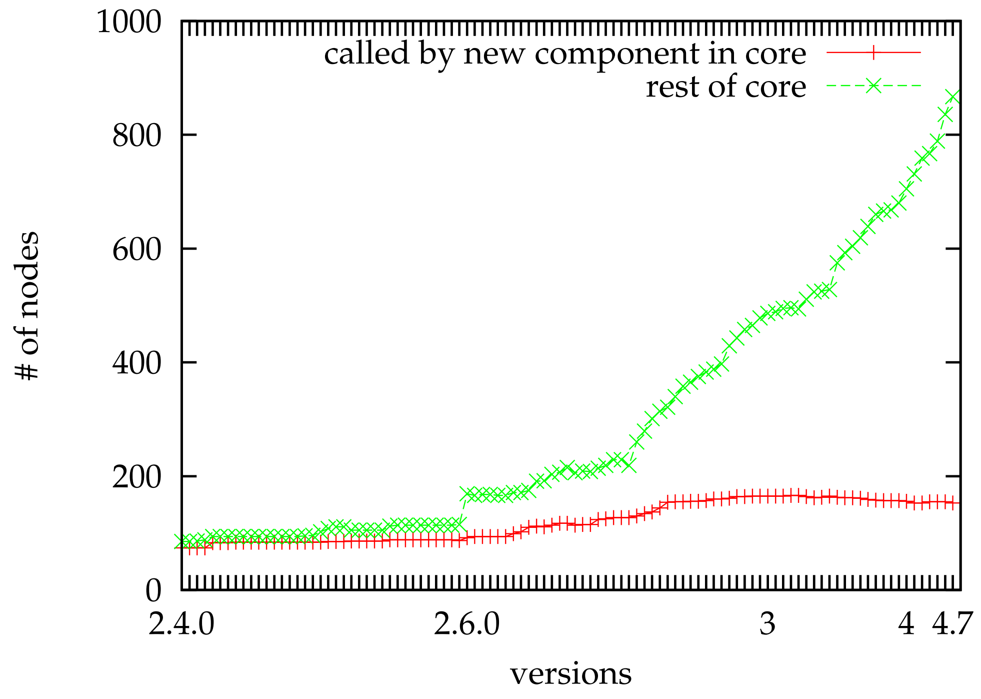


Figure 11: Changes of the number of nodes in core

5 Conclusion

In this thesis, we focused on the Linux kernel implementation, extract commonly used function group (a knot of bow-tie structure), and saw how the core function is used from functions other than core from the viewpoint of graph theory. As a result, Core was used by all components and found to play a fundamental role. Furthermore, by applying the above analysis to multiple versions of the Linux kernel, we analyzed the transition of core functions made by Linux kernel development from the viewpoint of dependency and size between functions, and organize the requirements for extraction and placement of core functions. Core does not change as development progresses, and core is always used when new components are added. On the other hand, it is speculated that ipv4 and irda are important rather than core when network functions are used from the outside of the Linux kernel. Also, the functions in core and ipv4 do not change so much as development progresses. From these facts, it is considered that the network function in the Linux kernel forms a bow-tie structure with a core as a knot.

Acknowledgements

This thesis would not accomplish without a lot of great supports of several people. I really appreciate Professor Masayuki Murata of Osaka University, for his invaluable advice, continuous support and encouragement. Furthermore, I show my deepest appreciation to Associate Professor Shin'ichi Arakawa of Osaka University. He devoted a great deal of time for me and gave me an excellent guideline of my research and considerable supports. This thesis could never be achived without his support. Also, I am deeply grateful to Assistant Professor Yuichi Ohsita and Assistant Professor Daichi Kominami of Osaka University, for helpful comments. Finally, I would like to thank all the members of Advanced Network Architecture Research Laboratory of Osaka University for their support.

References

[1] K. Ashton, “That ‘internet of things’ thing,” *RFiD Journal*, vol. 22, pp. 97–114, June 2009.

[2] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, “Internet of Things (IoT): A vision, architectural elements, and future directions,” *Future generation computer systems*, vol. 29, pp. 1645–1660, Sept. 2013.

[3] L. Atzori, A. Iera, and G. Morabito, “The internet of things: A survey,” *Computer networks*, vol. 54, pp. 2787–2805, Oct. 2010.

[4] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, “Internet of things: Vision, applications and research challenges,” *Ad Hoc Networks*, vol. 10, pp. 1497–1516, Sept. 2012.

[5] J. Pan, S. Paul, and R. Jain, “A survey of the research on future Internet architectures,” *IEEE Communications Magazine*, vol. 49, pp. 26–36, July 2011.

[6] A. Fischer, J. F. Botero, M. Till Beck, H. De Meer, and X. Hesselbach, “Virtual network embedding: A survey,” *IEEE Communications Surveys & Tutorials*, vol. 15, pp. 1888–1906, Feb. 2013.

[7] M.-K. Shin, K.-H. Nam, and H.-J. Kim, “Software-defined networking (SDN): A reference architecture and open APIs,” in *Proceedings of IEEE International Conference on ICT Convergence*, pp. 360–361, Oct. 2012.

[8] B. Partha, Z. Shuqiang, C. Pulak, L. Sang-Soo, L. J. Hyun, and M. Biswanath, “Software-defined optical networks (SDONs): A survey,” *Photonic Network Communications*, vol. 28, pp. 4–18, Aug. 2014.

[9] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog computing and its role in the internet of things,” in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pp. 13–16, Aug. 2012.

[10] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, “Fog computing: A platform for internet of things and analytics,” in *Big Data and Internet of Things: A Roadmap for Smart Environments*, pp. 169–186, Sept. 2014.

[11] S. Yi, C. Li, and Q. Li, "A survey of fog computing: concepts, applications and issues," in *Proceedings of Workshop on Mobile Big Data*, pp. 37–42, June 2015.

[12] A. Ahmed and E. Ahmed, "A survey on mobile edge computing," in *Proceedings of Intelligent Systems and Control (ISCO)*, pp. 1–8, Jan. 2016.

[13] NTT DOCOMO, INC., "DOCOMO Develops First NFV Technology for Multi-vendor EPC Software." https://www.nttdocomo.co.jp/english/info/media_center/pr/2016/0219_00.html. Accessed: 15 Dec. 2016.

[14] M. Gorman, "Codeviz: A call graph visualiser." Available at: `http://www.csn.ul.ie/~mel/projects/codeviz/`. Accessed: 1 Feb. 2015.

[15] M. Gorman, "Egypt - create call graph from GCC RTL dump." Available at: `http://www.gson.org/egypt/egypt.html`. Accessed: 1 Feb. 2015.

[16] "Linux Assembly HOWTO." Available at: `http://www.tldp.org/HOWTO/html_single/Assembly-HOWTO/`. Accessed: 1 Feb. 2017.

[17] Y. Gao, Z. Zheng, and F. Qin, "Analysis of Linux kernel as a complex network," *Chaos, Solitons & Fractals*, vol. 69, pp. 246–252, Dec. 2014.

[18] H. Wang, Z. Chen, G. Xiao, and Z. Zheng, "Network of networks in Linux operating system," *Physica A: Statistical Mechanics and its Applications*, vol. 447, pp. 520–526, Apr. 2016.

[19] L. Wang, P. Yu, Z. Wang, C. Yang, and Q. Ye, "On the evolution of Linux kernels: a complex network perspective," *Journal of software: Evolution and Process*, vol. 25, pp. 439–458, May 2013.

[20] T. Bu and D. Towsley, "On distinguishing between Internet power law topology generators," in *Proceedings of IEEE INFOCOM*, pp. 638–647, June 2002.

[21] Q. Chen, H. Chang, R. Govindan, S. Jamin, S. J. Shenker, and W. Willinger, "The origin of power laws in Internet topologies revisited," in *Proceedings of IEEE INFOCOM*, pp. 608–617, June 2002.

[22] M. Faloutsos, P. Faloutsos, and C. Faloutsos, “On power-law relationships of the Internet topology,” *ACM SIGCOMM Computer Communication Review*, vol. 29, pp. 251–262, Aug. 1999.

[23] C. Gkantsidis, M. Mihail, and A. Saberi, “Conductance and congestion in power law graphs,” *SIGMETRICS Performance Evaluation Review*, vol. 31, pp. 148–159, June 2003.

[24] H. Tangmunarunkit, R. Govindan, S. Jamin, S. Shenker, and W. Willinger, “Network topologies, power laws, and hierarchy,” *Computer Communication Review*, vol. 32, pp. 1–26, Jan. 2002.

[25] “Linux Kernel Archives.” `https://www.kernel.org/pub/`.

[26] T. Dreibholz, E. P. Rathgeb, I. Rüngeler, R. Seggelmann, M. Tüxen, and R. R. Stewart, “Stream control transmission protocol: Past, current, and future standardization activities,” *IEEE Communications Magazine*, vol. 49, pp. 82–88, Apr. 2011.

[27] K. A. Eriksen, I. Simonsen, S. Maslov, and K. Sneppen, “Modularity and Extreme Edges of the Internet,” *Physical Review Letters*, vol. 90, pp. 1–4, Apr. 2003.

[28] V. Blondel, J. Guillaume, R. Lambiotte, and E. Mech, “Fast unfolding of communities in large networks,” *Journal of Statistical Mechanics*, pp. 1–12, July 2008.

[29] C. Y. Baldwin and K. B. Clark, *Design Rules: The Power of Modularity*. MIT Press, 1999.