Evolution of Functional Core in Network-related Function Calls during Linux Kernel Development

Shin'ichi Arakawa*, Hirotaka Miyakawa*, Tetsuya Takine† and Masayuki Murata*
* Graduate School of Information Science and Technology, Osaka University, Japan
{arakawa, h-miyakawa, murata}@ist.osaka-u.ac.jp
† Graduate School of Engineering, Osaka University, Japan
takine@comm.eng.osaka-u.ac.jp

Abstract—**Recently, network function virtualization has been focused on achieving a flexible deployment of networking services. Despite the fact that the heart of network function virtualization is its software implementation, there are fewer studies on how network functions are implemented as software. In this study, we investigate the evolution of functional connectivity in network functions using an implementation of Internet protocol suite in the Linux kernel. We constructed a call graph for the Linux kernel and analyzed the change of connectivity between the protocol components based on the directory structure of Linux kernel. Our results on the connectivity analysis show that new sub-directories appears for new emerging technologies, such as "bluetooth" and "sctp", and they rely mostly on the function of sub-directories "core" and "ipv4". Since the number of functions in sub-directories "core" and "ipv4" is increasing, we defined a functional core for the call graph to see whether there is a core part which has lower variability during the kernel development or not. The result shows that the functional core consists of mostly 50-70 functions and has lower variability comparing with the increase of a number of network-related functions during the kernel development.**

I. Introduction

The Internet has been more and more important tool for our daily life as smartphones and tablet computers become widely used. Various Internet services have emerged along with the diversity of devices and functional/performance requirements. In contrast, IP (Internet Protocol) has been centered on the protocol suite of the Internet and was not so changed in its functionality during the history of the Internet.

It is widely accepted that the protocol suite of the Internet is IP-centric and resembles an hourglass. That is, various kinds of applications use networking services through a socket interface which provides TCP/UDP as the transport layer and IP as the network layer. Also, various kinds of bit-transport systems can be used by IP through interfaces of the data link layer such as Ethernet or IEEE 802.11's media access control. From the application development point of view, IP-centric protocol suite nicely hides the detail of network infrastructure thanks to the socket interface. On the other hand, because of the socket interface, the application developers will face with difficulties when they want to change networking functions such as routing and/or addressing. From the networking point of view, deploying new networking services at the network layer is not easy because most of the networking services are now implemented using the hardware.

Recently, network function virtualization has been focused on achieving a flexible deployment of networking services. Virtualization is a technique to prepare virtual machines on top of a hardware. Then, software-implemented network functions are executed on top of the virtual machine. Network operators can easily deploy a new network function on the network infrastructure by, e.g., slicing a new virtual machine and by executing the software-implemented network function on the sliced virtual machine.

Despite the fact that the heart of network function virtualization is its software implementation, there are fewer studies on how network functions are implemented as software. One of the reasons may be that the network function virtualization is at the early stage of standardization. Another reason may be that studies of a software implementation are not the main scope of networking research community. Actually, Linux kernel, which includes software implementation of the protocol suite, has been analyzed in a software-engineering research community where an analysis of software quality and bug prediction are main concerns. Gao et al. [1] generated the call graph of the Linux kernel and analyzed the change of the Linux kernel using some metrics such as degree distribution and average path length. In the study, they reveal that the indegree distribution follows power-law whereas the outdegree distribution follows an exponential distribution. They demonstrate that the Linux kernel is highly modularized and the failure of the nodes with large in-degree do more damage on the entire system. Wang et al [2] investigate the degree distribution of each component by regarding a directory as a component. Koon et al. [3] compared the transcriptional regulatory network of a bacterium (Escherichia coli) with the network that represents function call relationship of the whole Linux kernel and showed that functions are heavily reused in the Linux kernel. However, both of works focuses on the analysis of entire Linux kernel.

In this paper, using an implementation of Internet protocol suite in the Linux kernel, we investigate the evolution of connectivity between network functions. More specifically, we construct a graph for each kernel version by regarding a node as a function call and a link as a callee-caller relationship of function calls. Then, changes of topological characteristic are investigated. We mainly focus on the functional core of the network-related implementation, which is defined as a group of functions that maximally handle the information processing requests from remaining functions.

The rest of this paper is organized as follows. Section 2 explains a method to obtain a graph representation of Linux kernel implementation. In Section 3, we investigate the way of evolution of network-related implementation in Linux kernel based on the directory information. Section 4 define the functional core in the Linux kernel implementation and investigate the evolution of the functional core during the kernel development. Section 5 concludes this study.

II. Graph representation of Linux kernel implementation

A. Obtaining a call graph for Linux kernel

The call graph is a directed graph with functions as nodes and function calls as edges. When a function (caller) calls another function (callee), the edge connects these two nodes. CodeViz [4] is one of the tools for generating a call graph from program code and is commonly used for visualizing program code to understand the structure of the software implementation. However, we do not use the CodeViz for obtaining a call graph for Linux kernel. One of reason is that we were faced with a problem that functions that are declared at different files but has the same name are regarded as the same node in the call graph generated by the CodeViz. Therefore, we use GCC's debugging option (dump-rtl-expand) to obtain a Register Transfer Language code for each source .c file. RTL is the intermediate language used for exchanging the front-end, which performs lexial analysis and syntax analysis, and the back-end, which optimize the program code and generate binary code. Since the RTL code is nicely formatted and generated each source file, we developed a software tool for analyzing the RTL code. We used a gcc version 6.1.1 (20160621) to compile the Linux kernel from 2.4.0 to 4.7. The configuration options for the compilation was set to the default settings of Fedora 24. Note that recent gcc sometimes fails to compile the old Linux kernel due to the change of the specification of gcc. In this case, we give a little modification to the Linux kernel such that the recent gcc can compile the Linux kernel.

Even after we have obtained RTL codes, we are still facing a problem that we cannot distinguish two functions that have the same name. For a caller-callee relationship, the caller function is easily identified since the RTL code is generated for each source file. The problem is the identification of the callee function: which function is actually called by the caller function? GCC compiler uses the linker program in actual to identify the body of the callee function, but it was difficult for us to follow the behavior of linker program. Instead, we apply following rules in sequence to identify the location of the callee function.

1) When the caller function and callee function is defined in the same file, the callee function located at the same file is called.

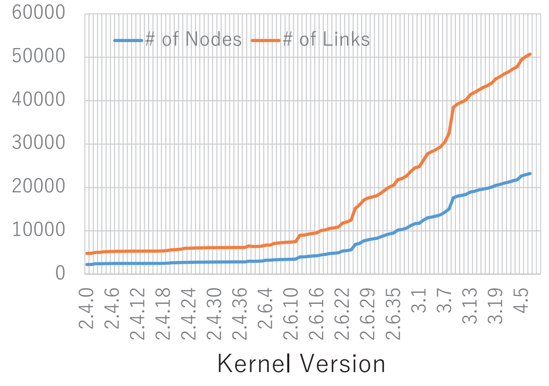

Fig. 1. Changes of numbers of nodes and links of network-related functions from Linux kernel 2.4.0 to 4.7

2) When the callee function is defined only in one file in the entire Linux kernel, the callee function defined at the file is called.
3) When the name of the callee function appears at the several files in the entire Linux kernel, the callee function located at the closest directory to the caller function is called.
4) When the name of the callee function appears at the declaration of assembler file, the callee function defined at the assembler file is called.

There are mostly 100 callee functions that cannot be identified by applying the above rules. These functions include, for example, `acpi_pci_link_exit`, `acpi_ec_exit`, `do_suspend_lowlevel_s4bios`, and so on. Most of these functions are related to the BIOS-level function call and are not related to the networking function, so we will ignore these function calls for further analysis. From Linux kernel 2.4.0 to 4.7, the number of nodes was increased from 8,806 to 164,945 and the number of links was increased from 25,813 to 602,149.

B. Obtaining a call graph for network related functions in Linux kernel

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 useful to extract. Files of 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", "ipv4", and others. In this paper, we regard a function which is defined at the files under the branch of "net" directory as the network-related function. In addition, we regard a function which is defined at the files under the branch of "drivers" directory and calls or be called by the function under the branch of "net" directory as the network-related function. The "drivers" directory gathers the hardware-related function calls for not only the network drivers but also storage drivers and graphic-related drivers. That is, a part of functions defined under the branch of “drivers” directory is supposed to be a network-related function. So, we consider that the function under the branch of “drivers” directory is a network-related function when it is related to the functions under the branch of “net” directory.

Figure 1 shows the changes numbers of network-related functions and links that connect them from Linux kernel 2.4.0 to 4.7. 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 [5]. The number of network-related functions was increased from 2,281 to 23,219 and the number of links was increased from 4,834 to 50,703. Numbers of nodes and links have increased as development has progressed since new function calls are added without deleting old function calls. From version 2.4.0 to version 3, numbers of nodes and links drastically increased due mainly to IPSec and IPv6 support. Note that the implementation of IPSec and IPv6 exists before the version 2.4.0, but it was EXPERIMENTAL. In this paper, all of the results are based on the version at which the default configuration option is activated. After version 3.0.0, numbers of nodes and links are not drastically increased except from version 3.7 to 3.9 where the number of nodes increased from 14,331 to 17,631 and number of links significantly increased from 30,477 to 38,445. We check the name of function calls in detail and conclude that Stream Control Transmission Protocol (SCTP) [6] was activated by default from version 3.8.0. The increase of a number of links is faster than the increase of the number of nodes, which means that a function is being actively reused.

III. Evolution of network-related implementation in Linux kernel

In this section, we investigate changes in connectivity in network-related functions during the development of Linux kernel. We use our own software tool (See Sec. II for detail) to obtain network-related functions and their connectivity in the Linux kernel from version 2.4.0 in Jan. 2001 to version 4.7 in Jul. 2016.

A. Analysis of connectivity between directories

We first investigate the relation between the network functions based on the (sub-)directories that the functions are belonging. As we mentioned above, files related to network functions are gathered in the “net” directory whose sub-directories are also grouped into more specific functions such as “ethernet”, “ipv4”, and others. Figure 2 shows the connectivity between sub-directories in the Linux kernels 3.0 and 4.7. Here, the node is the sub-directory and the link between two nodes is constructed when one or more functions defined at files in a sub-directory calls one or more functions defined at files in another sub-directory. In the figure, the size

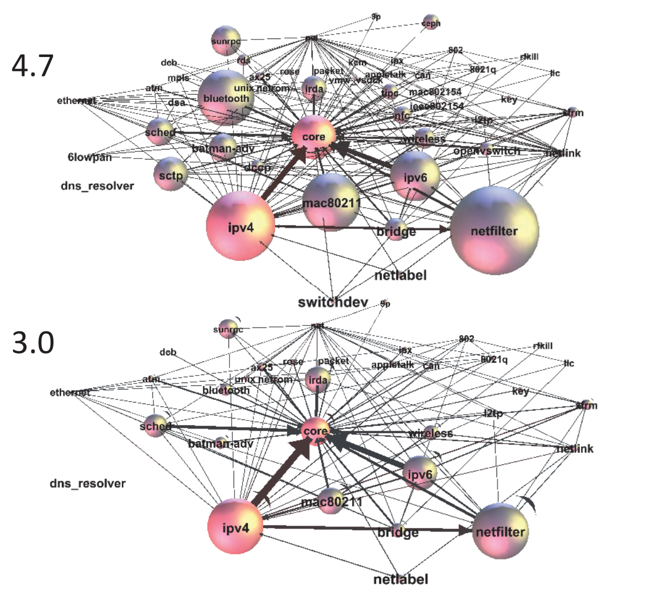

Fig. 2. Connectivity between sub-directories of the “net” directory at Linux kernel 3.0 and 4.7

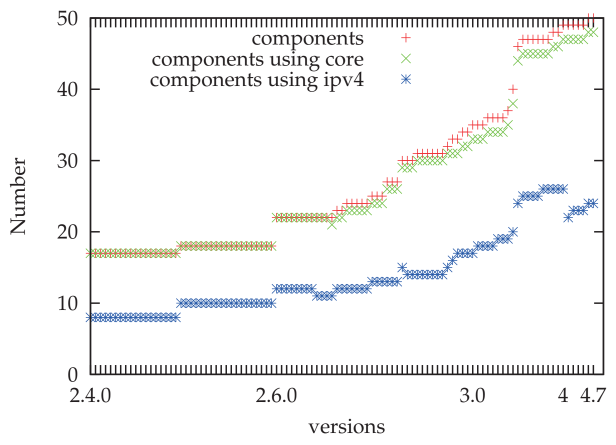

Fig. 3. Number of sub-directories (protocol components) connected to “core” and “ipv4” from 2.4.0 to 4.7.

of the node expresses the number of calls in the component, and the thickness of the link expresses the number of calls between the protocol components. The figure clearly shows that functions have been increased for each sub-directory. The network-related functions for new emerging technologies, such as “bluetooth”, “sctp”, and “openvsswitch”, has appeared or drastically increased from 3.0 to 4.7. An interesting observation can be made for sub-directory “core” and “ipv4”: most of the newly added sub-directory are connected to the sub-directory “core” and 60% of them are connected to the sub-directory “ipv4”

Figure 3 shows the changes of the number of sub-directories and changes of the number of sub-directories connected to “ipv4” and “core”. The sub-directories that use “core” and “ipv4” are increasing as the total number of sub-directories

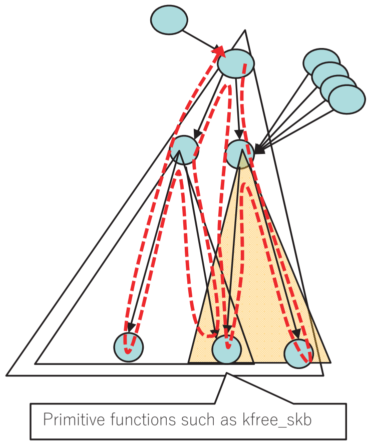

Fig. 4. Relation between a call graph and a flow of processing.

increases. “core” and “ipv4” are almost always used from new sub-directory is added. Thus, the components “coreh and “ipv4h play a key role in providing network functions.

B. Weight-based analysis

In the previous section, we analyzed the connectivity between network-related functions based on the directories that they are belonging. In the analysis, we assume that the weight of connection between functions is always 1.0. However, when we want to know the importance of network functions, it is necessary to consider the frequency of function calls for each link. To explain this, we show the relation between a call graph and a flow of processing in Figure 4. In the figure, nodes and solid lines form a call graph, that is, a node represents a function and the link with a solid line represents a caller-callee relationship between functions. The triangle represents a unit of information processing, i.e., the function call of the node will be completed after the all of the nodes inside the triangle are completed. Thus, the dashed-line depicts the flow of processing when the function of the top-node in the biggest triangle is called. This figure suggests that the link in the call graph can be called many times in an actual usage, and therefore, we have to define the weight of links to reflect the frequency of function calls for each link. One of the ways to define the weight of links is to run the Linux kernel in the actual environment and then measure the number of function calls directly. However, it is difficult to define the “actual” environment. Therefore, instead of taking a way to run the Linux kernel, we consider the graph-based definition for the link weight. Looking at the Figure 4, we notice that a function call is triggered by nodes whose indegree is 0. Thus, we define the weight of links by counting up the appearance of each link during the flow of processing triggered from indegree-zero nodes.

Figure 5 shows the number of internal links for each sub-directory at the Linux kernel 4.7. Here, the internal link for a sub-directory means that its caller function and callee function belongs to the same sub-directory. We also show

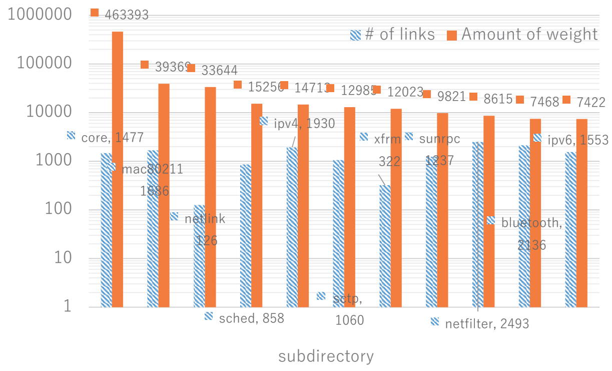

Fig. 5. Total weight of internal links for each sub-directory (selected, LKM v4.7).

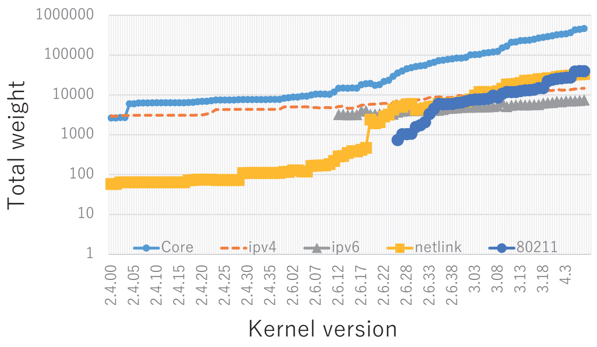

Fig. 6. Changes of total weight for each sub-directory.

the sum of weight for internal links in the figure. The sub-directory “core” has a lot of highly-used links internally, and its value is higher than other sub-directory by one order of magnitude. The importance of sub-directory “netlink” is also drastically changed. The function call defined in the sub-directory “netlink” is only 126, but they are called 33,644 times. We finally check the evolution of total weight for each sub-directory from Linux kernel 2.4.0 to 4.7. Figure 6 shows the total weight of links in each sub-directory dependent on the Linux kernel version. The total weight of sub-directory “80211” (for wifi communication), “netlink”, and “core’ increases continuously, but the that of sub-directory “ipv4” and “ipv6” does not increase so much.

Next, we investigate the relation between sub-directories under the directory “net”. Our results on the connectivity analysis show that new sub-directories appears for new emerging technologies, such as “bluetooth” and “sctp”, and they rely mostly on the function of sub-directories “core” and “ipv4”. However, our analysis based on the link weight reveals that the importance of the sub-directory “ipv4” is not so high comparing with the sub-directory “core’. The sub-directory “netlink” is important rather than the “ipv4” and “ipv6”. As the name of directory “core” indicate, the sub-directory “core” plays a central role for network functions. A question is whether all of the functions under the sub-directory “core” is important or not. Another question is whether the size of important functions in the directory “net” increases or not. In the next section, we develop a method to reveal the functional core in the network-related implementation and presents the way of evolution of the functional core during the kernel development.

IV. Evolution of Functional Core of the Network-related Functions

In this section, we extract the functional core, which defined as a set of function that plays a central role in information processing, of the network-related functions through a “Core-Periphery” concept [7]. The core-periphery concept interpret a system into a core part, which has lower variability and is efficient, and a periphery part, which has higher variability to absorb environmental changes. Our interest on the Linux kernel implementation is whether there is a core part which has lower variability during the kernel development or not.

Defining the core part in the Linux kernel is not an easy task because the lower variability is not a sufficient condition to define the core. For example, functions of old networking technology that are not maintained now are unchanged during the recent kernel development. Avin et al [8] presented an axiom-based definition to separate a social network into the core and periphery part. According to the [8], the core part satisfies following four conditions; A) Dominance where the total of weight of links connecting core part and periphery part is greater than that of links inside the periphery part, B) Robustness where the total of weight of links inside the core part is greater than that of links connecting core part and periphery part, C) Compactness where the size of the core part is minimum, D) Density, where the density of the core part is sufficiently high. However, in this paper, we use an another definition of the functional core intended for the Linux kernel. The functional core is defined as the set of functions such that the total weight of links connecting the functional core and the remaining part is maximized. As we have shown in Figure 4, the weight of links is defined as the number of function calls. Thus, the functional core by our definition represents the set of highly called functions from the remaining part. Figure 7 show the changes of functional core from Linux kernel 2.4.0 to 4.7. The figure shows that the functional core consists of mostly 50-70 functions and has lower variability comparing with the increase of a number of network-related functions during the kernel development.

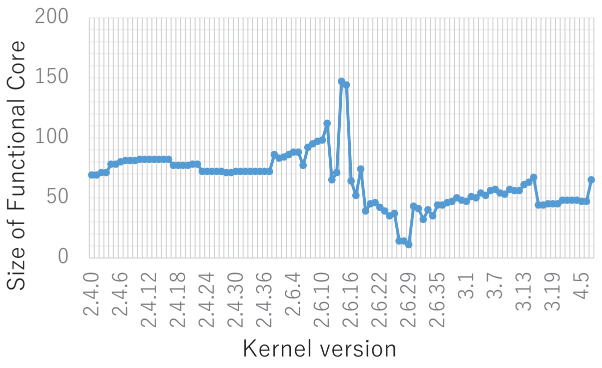

Fig. 7. Changes of the size of functional core.

V. Conclusion

In this study, we investigated the evolution of functional connectivity in network functions using an implementation of Internet protocol suite in the Linux kernel. We constructed a call graph for the Linux kernel and analyzed the change of connectivity between the protocol components based on the directory structure of Linux kernel. Our results on the connectivity analysis show that new sub-directories appears for new emerging technologies, such as “bluetooth” and “sctp”, and they rely mostly on the function of sub-directories “core” and “ipv4”. However, our analiysis based on the link weight reveals that the importance of the sub-directory “ipv4” is not so high comparing with the sub-directory “core’. Then, we defined a functional core for the call graph to see whether there is a core part which has lower variability during the kernel development or not. The result shows that the functional core consists of mostly 50-70 functions and has lower variability comparing with the increase of a number of network-related functions during the kernel development.

The analysis in this paper was based on the analysis of source codes and actual behavior of the program is not reflected. Our future work is to analyze the inter-dependency of network functions when the Linux kernel works for some networking purpose.

Acknowledgement

This work was supported by National Institute of Information and Communications Technology (NICT) in Japan.

References

[1] Y. Gao, Z. Zheng, and F. Qin, “Analysis of linux kernel as a complex network,” *Chaos, Solitons & Fractals*, vol. 69, pp. 246–252, Nov. 2014.

[2] 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.

[3] K.-K. Yan, G. Fang, N. Bhardwaj, R. P. Alexandera, and M. Gersteinb, “Comparing genomes to computer operating systems in terms of the topology and evolution of their regulatory control networks,” *Proceedings of the National Academy of Sciences*, vol. 107, no. 20, pp. 9186–9191, May 2010.

[4] M. Gorman, “Codeviz: A callgraph visualiser,” Available at: http://www.csn.ul.ie/ mel/projects/codeviz/, accessed: 1 Feb. 2015.

[5] “The Linux Kernel Archives,” Available at: http://www.kernel.org, accessed: 1 Feb. 2017.

[6] 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, no. 4, pp. 82–88, Apr. 2011.

[7] P. Csemely, A. London, L. Wu, and B. Uzzi, “Structure and dynamics of core/periphery networks,” *Journal of Complex Networks*, vol. 1, no. 1, pp. 93–123, Oct. 2013.

[8] C. Avin, Z. Lotker, D. Peleg, Y. A. Pignolet, and I. Turkel, “Core-periphery in networks: An axiomatic approach,” *arXiv preprint arXiv:1411.2242*, Nov. 2014.