Pamukkale Üniversitesi Mühendislik Bilimleri Dergisi

Pamukkale University Journal of Engineering Sciences

Analysis of function-call graphs of open-source software systems using complex network analysis

Karmaşık ağ analizi kullanılarak açık-kaynak yazılım sistemlerinin fonksiyon-çağırma graflarının analizi

Volkan TUNALI[1]*, Mehmet Ali Aksoy TÜYSÜZ[2]

[1,2]Department of Software Engineering, Faculty of Engineering and Natural Sciences, Maltepe University, İstanbul, Turkey.
volkan.tunali@gmail.com, mehmetaliaksoy@gmail.com

Received/Geliş Tarihi: 03.08.2018 Revision/Düzeltme Tarihi: 04.03.2019 doi: 10.5505/pajes.2019.63239
Accepted/Kabul Tarihi: 10.05.2019 Research Article/Araştırma Makalesi

Abstract

Software systems are usually designed in a modular and hierarchical fashion, where functional responsibility of a system is decomposed into multiple functional software elements optimally such as subsystems, modules, packages, classes, methods, and functions. These elements are coupled with each other with some kind of dependency relationships to some degree, and their interactions naturally form a graph or network structure. In this study, we generated the static function-call graphs of several open-source software systems, where functions were the most basic type of interacting elements calling each other. Then, we analyzed the call graphs both visually and topologically using the techniques of complex network analysis. We found the call graphs to reveal scale-free and small-world network properties similar to the findings of the previous studies. In addition, we identified the most central and important functions in each call-graph using several centrality measures. We also performed community analysis and found that the call graphs exhibited a tendency to form communities. Finally, we showed that analysis of static function-call graphs of software systems through complex network analysis has the potential to reveal useful information about them.

Keywords: Fuction-call graph, Open-source software, Complex network analysis, Network centrality, Network science.

Öz

Yazılım sistemleri genellikle sistemin işlevsel sorumluluğunun optimal bir şekilde altsistemler, modüller, paketler, sınıflar, metodlar, ve fonksiyonlar gibi çok sayıda işlevsel yazılım elemanına ayrıştırıldığı, modüler ve hiyerarşik bir biçimde tasarlanırlar. Bu elemanlar birbirleriyle çeşitli ilişki türleri ile bağlıdırlar ve bunların etkileşimleri doğal olarak bir graf veya ağ yapısı oluşturur. Bu çalışmada, etkileşim halindeki en temel eleman türü olarak birbirini çağıran fonksiyonları dikkate alarak, çeşitli açık-kaynak yazılım sistemlerinin statik fonksiyon-çağırma graflarını oluşturduk. Ardından, karmaşık ağ analizi teknikleri kullanarak, çağırma graflarını hem görsel hem de topolojik olarak analiz ettik. Daha önceki çalışmaların bulgularına benzer olarak, grafların ölçekten-bağımsız ve küçük-dünya ağı özellikleri sergilediklerini gördük. Ek olarak, çeşitli merkezîlik ölçütleri kullanarak, her bir çağırma grafındaki en merkezi ve önemli fonksiyonları tespit ettik. Ayrıca, topluluk analizi gerçekleştirdik ve çağırma graflarının topluluk oluşturma eğilimi gösterdiğini bulduk. Son olarak, yazılım sistemlerinin statik fonksiyon-çağırma graflarının karmaşık ağ analizi yoluyla analizinin, sistemlerle ilgili yararlı bilgiler sağlama potansiyeli olduğunu gösterdik.

Anahtar kelimeler: Fonksiyon-çağırma grafı, Açık-kaynak yazılım, Karmaşık ağ analizi, Ağ merkezîliği, Ağ bilimi.

1 Introduction

Computer software development is a complicated process with several distinct activities like requirements analysis and development, architectural and detailed design, coding and debugging, testing, and maintenance. Real-world problems become larger and more complex than ever with intricate and a large number of interacting entities. This complexity manifests itself in the development of software solutions that address these real-world problems, and it makes the whole development process essentially difficult. Therefore, concept of complexity in software development cannot be overlooked. McConnell emphasizes this as "*Managing complexity is the most important technical topic in software development. In my view, it's so important, that Software's Primary Technical Imperative has to be managing complexity.*" [1]. The main goal of software architecture and design techniques is to deal with inherent complexity by dividing a complex problem into simpler and more managable pieces. Thus, software systems are usually designed in such a modular and hierarchical fashion that functional responsibility of a system is decomposed into multiple functional software elements optimally such as subsystems, modules, packages, classes, methods, and functions. Obviously, these elements are not independent of each other. Instead, they are coupled with each other with some kind of dependency relationships to some degree, and their interactions naturally form a graph or network structure.

A widely used type of software graphs is the function-call graph, where nodes represent the functions in a software system, and edges represent the action of function call between the functions. Self-loops in the graph indicate recursive function calls. Call graphs can be both static and dynamic. Static call graphs are generated by analyzing the source code structure. Dynamic call graphs, on the other hand, are captured during the execution of the software system. Call graphs are utilized for several purposes, such as program understanding, compiler optimization, etc.

Complex networks have been actively studied for about two decades by researchers from the emerging field of network science [2]. Software function-call graphs are no exception to this. In this research, we focused on four different open-source systems, all written in C programming language, namely

*Corresponding author/Yazışılan Yazar

GNOME Library, GNU Emacs, Git, and Linux OS Kernel. We generated four static function-call graphs from the source code files of these systems. Then, we used complex network analysis tools and techniques to explore and analyze the call graphs. We compared the topological properties of the networks with each other and with the findings of the previous studies. Besides, we revealed the most important functions in each system through centrality analysis.

The paper proceeds as follows. Section 2 discusses related work on the complex network analysis of software call graphs. Section 3 gives the details of the material and the method used for the analysis. Section 4 presents and discusses the findings. Section 5 then concludes the paper while also recommending directions for future work.

2 Related work

There are several notable studies in the literature that analyzed software networks of different domains and sizes.

In [3], software collaboration graphs of several open-source software systems written in C and C++ programming languages were analyzed, and they were found to reveal scale-free and small-world properties observed in many other systems. In addition, several topological measures of these networks were examined with respect to software engineering practices.

In a comparative study, the transcriptional regulatory network of E. Coli bacterium and the call graph of Linux operating system written in C programming language were analyzed [4]. Topological analysis results of the study showed that both networks had a fundemantally hierarchical layout while their organizations led to the robustness of the biological system and the cost effectiveness via reusability of the software system.

In another study, function-call graph of the open-source Celestia application written in C++ programming language was examined and it was shown that the graph exhibited scale-free and small-world properties [5].

In [6], function-call graphs of ten open-source Java programs were analyzed. The study showed that the call graphs exhibited scale-free and small-world properties. In addition, the study found that functions with high out-degree provided high-level business functions, and those with high in-degree provided low-level supportive functions.

In a study by Qu et al., by using four community detection algorithms in the analysis of 10 open-source Java programs, it was shown that static call graphs usually present significant community structures [7]. Moreover, two class cohesion metrics were proposed in that study.

In a more recent study, topological properties of the call graph of the Hibernate software, written in Java programming language, were analyzed [8]. Similar to the previous studies, the analysis revealed that the network showed scale-free and small-world properties.

3 Materials and method

Our research methodology consists of three distinct phases: data gathering, network modeling and construction, and complex network analysis. Details of each phase are given under their respective sub-sections in this section.

3.1 Data gathering

Data for this study were collected as source code files written in C programming language from the github repositories of the projects GNOME Library 3.29.3 [9], GNU Emacs 27.0.50 [10], Git 2.19 [11], and Linux kernel 4.18.0-rc6 [12]. There were several common features of these projects that influenced our decision to choose them for this research. First, all of them were written purely in C programming language. Second, they were all open-source software, which promotes collaboration and sharing by allowing other people to make modifications to source code and incorporate those changes into their own projects. Third, these projects had been contributed to by people all around the world usually without under control of a central body, preserving their internal code quality.

3.2 Network modeling and construction

A function-call graph is a directed graph, where nodes represent the functions in a software system, and edges represent the action of function call between the functions. Self-loops in the graph indicate recursive function calls. Call graphs can be both static and dynamic. Static call graphs are generated by analyzing the source code structure. Dynamic call graphs, on the other hand, are captured during the execution of the software system. In this research, our focus was on static call graphs.

We used pycflow2dot 0.2 [13] to generate function-call graphs of the projects. pycflow2dot is a Python script that essentially uses GNU cflow [14] to generate call graphs. GNU cflow analyzes a collection of C source files and prints the control flow within the program. pycflow2dot generates graphs in Graphviz dot format [15], which is a common graph format supported by many network analysis tools.

3.3 Complex network analysis

Complex Network Analysis is a set of techniques that essentially studies the relationship between the structure and the function of large and complex networks where nodes represent any kind of entities, and edges represent any kind of relationships between the entities [16]. It is based on theories and methods from several disciplines including mathematics, physics, computer science, statistics, and sociology. A network analysis study usually consists of the following steps after a network representation of the complex phenomenon of interest is obtained. First, the complex network is visualized and patterns are searched for visually. Then, the structural analysis is performed to understand the general characteristics of the network as a whole. For example, structural measures like degree distribution of nodes, clustering coefficient, average path length, and diameter give us important information about the network. Third phase is the centrality analysis, where structurally the most central and important nodes are identified using appropriate centrality measures like degree, closeness, betweenness, and PageRank centrality. A final analysis called community analysis is performed to reveal the transitive relationships between nodes to detect dense groups of nodes called clusters in the network [17].

In this study, we used Cytoscape 3.6.1 [18] and Gephi 0.9.1 [19] to visualize and analyze the function-call graphs. Cytoscape is a general-purpose, open-source software environment specifically developed for the large scale integration of molecular interaction network data. However, it can be used to analyze any type of network owing to its extensive analysis capabilities and plugin-based extensibility. Gephi is a well-known, open-source free network visualization tool with capabilities to calculate centrality, clustering, network diameter, and other metrics.

4 Findings and discussion

Using the above mentioned network analysis tools and techniques, we analyzed the function-call graphs in four distinct phases. First, we visualized the graphs to see their general structure and layout. Then, we performed structural network analysis. Next, we calculated the centralities of the nodes. Finally, we observed the community structures embedded in the graphs.

4.1 Visual analysis

A complex network analysis usually begins with the visualization of the network using an appropriate layout algorithm that enables one to see the organization of the nodes and their relationships with each other as clearly as possible. For this reason, we visualized the call graphs using Yifan Hu layout algorithm available in Gephi [20]. Yifan Hu is a fast and effective force-directed layout algorithm that usually provides a clear view of the network. The visualizations of the call graphs are shown in Figure 1. The nodes were colored depending on the modularity class (community) they belonged to. Details of the community analysis is given in Section 4.4. In addition, the nodes were resized in proportion to their in-degrees.

4.2 Topological analysis

4.2.1 General topological measures

We obtained the general topological measures of the call graphs using Cytoscape, such as density, clustering coefficient, diameter, centralization, and characteristic path length. All these measures are presented in Table 1.

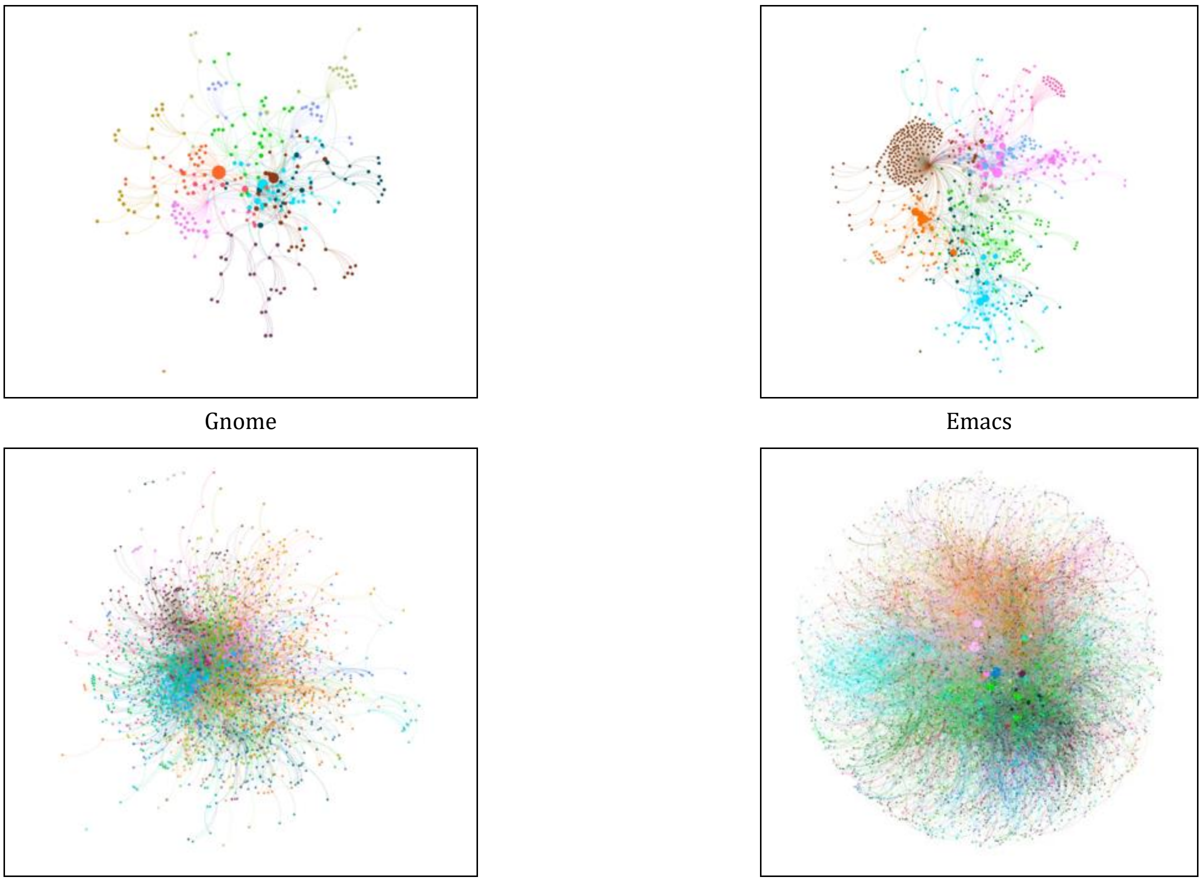


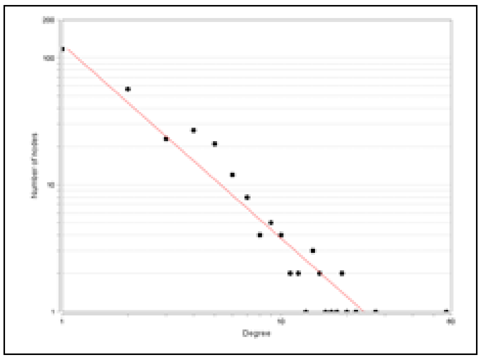
Figure 1. Visualizations of the call graphs.

Table 1. Topological measures of the call graphs.

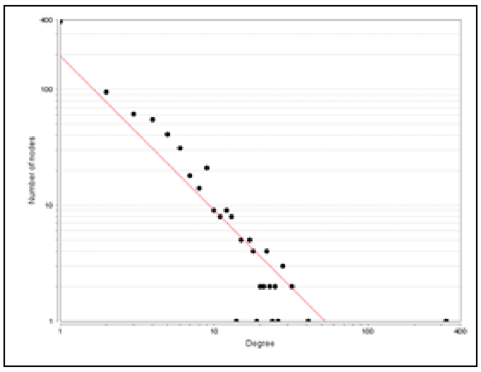
Measure	Gnome	Emacs	Git	Linux
Nodes	297	793	2,374	7,736
Edges	552	1,640	7,256	19,959
Clustering coefficient	0.076	0.082	0.070	0.054
Characteristic path length	3.907	3.593	3.984	4.855
Density	0.013	0.005	0.003	0.001
Connected components	1	1	7	167
Diameter	8	9	9	13
Radius	5	5	5	7
Avg. num. of neighbors	3.717	4.124	6.094	5.155
Centralization	0.181	0.404	0.070	0.033

4.2.2 Scale-free analysis

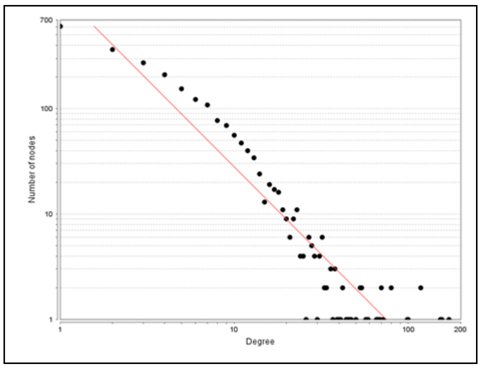
We obtained the degree distribution charts shown in Figure 2 and fit the distributions with Power Law distribution using Cytoscape. Table 2 shows the scaling factors for Power Law distributons of the graphs.



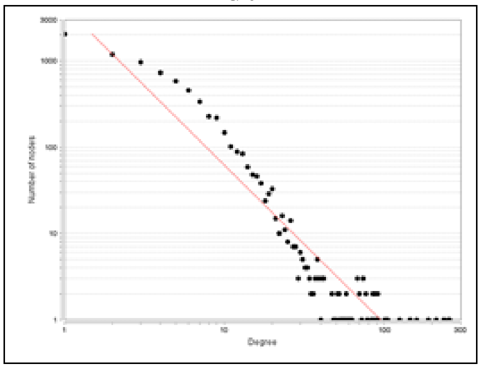
Gnome



Emacs



Git



Linux kernel

Figure 2. Degree distribution charts of the call graphs.

Table 2. Scaling factors for Power Law distributions.

Gnome	Emacs	Git	Linux
1.528	1.325	1.657	1.834

Visual inspection of the degree distribution charts and the values of scaling factors suggest that the graphs exhibited Power Law degree distribution, which is the distribution usually observed in most real-world networks regardless of the type and size of the network [17]. This distribution indicates that most of the nodes have a relatively low degree (number of neighbors) while a few nodes have a very high degree. High number of neighbors (or degree) can be attributed to the popularity/utility of the node (the corresponding function). We showed that the function-call graphs of different size had the scale-free property [21].

4.2.3 Small-world analysis

In addition to being scale-free, most real-world networks are known to be small-world, meaning that they show a higher clustering than random networks and they have a very short average path length regardless of their size [22].

In order to check the existence of small-world characteristics in the call graphs, we synthetically generated 10 different Erdös-Rényi [23] random graphs for each call graph with the same number of nodes and edges as each one using the network randomizer plugin in Cytoscape. Then, we compared the mean values of the structural analysis results of these corresponding random graphs to the results of the call graphs. Tables 3 and 4 show the comparisons.

Table 3. Clustering coefficient comparison.

Graph	Gnome	Emacs	Git	Linux
Original	0.076	0.082	0.070	0.054
Random	0.022	0.006	0.002	0.001

Table 4. Characteristic path length comparison.

Graph	Gnome	Emacs	Git	Linux
Original	3.907	3.593	3.984	4.855
Random	4.477	4.785	4.505	5.634

For all the call graphs, corresponding random graphs had much lower clustering coefficient values as expected. Moreover, characteristic path lengths of the corresponding random graphs were very close considering the size of the graphs. These findings were in concordance with the general expectation as it is a well-known fact that random networks do not exhibit the high clustering of real-world networks but short characteristic path length is a common behavior of both real-world and random networks. As a result, we showed that the function-call graphs presented small-world characteristics. Additionally, the degree distributions of the random graphs were far from Power Law; they were Poisson actually as expected as a result of random edge addition between nodes. When compared to the random graphs in terms of these three characteristic properties, we finally inferred that the software function-call graphs under study were far from randomness. This is not surprising because software developers do not decompose a software system into modules randomly and do not distribute function calls all over the code at random. Instead, they decide to divide a unit into subunits with a kind of optimization and reusability approach.

4.3 Centrality analysis

In this phase of the analysis, we calculated in-degree, out-degree, betweenness, and PageRank centralities of the nodes in the call graphs using Gephi. These are the most commonly used centrality measures that usually give useful insights about the relative importance of nodes in directed networks. The higher the centrality value, the more central a node is.

In-degree of a node (function) in call graphs is the number of other functions that directly call this function, also known as fan-in. Out-degree of a function is the number of other functions that this function directly calls, also known as fan-out. If a function has a high betweenness centrality value, it can be ragarded that this function acts as a middle layer between higher and lower abstraction layers in the call hierarchy. Functions with relatively high PageRank centrality values are the functions that are called by many other functions that are themselves are also called by high number of functions. This centrality value with a kind of recursive definition can be highly useful for identifying the important functions. Tables 5-8 present top 10 functions in the call graphs ranked by their in-degree, out-degree, betweenness, and PageRank centralities.

4.4 Community analysis

In this final phase of the complex network analysis, we used Gephi to identify the dense groups of nodes called communities (or clusters) in the call graphs. In Gephi, community analysis was performed with the Modularity operation, which is based on a technique called Modularity Optimization [24]. We used this operation with its only parameter *resolution* set to its default value of 1.0. Identified communities can be seen with distinct colors of nodes in the graph visualizations in Figure 1. Clusters of functions may be interpreted as an indication of modular and hierarchical organization of the units of the software systems of interest.

Table 5. Top 10 functions of Gnome by their centralities.

Rank	In-Degree	Out-Degree	Betweenness	PageRank
1	g_return_val_if_fail	main	collect_locales	g_return_val_if_fail
2	g_free	setup_seccomp	add_locale	g_assert
3	g_strdup	script_exec_new	parse_rules	g_free
4	g_hash_table_lookup	add_locale	collect_locales_from_directory	g_strdup
5	g_str_equal	parse_rules	ensure_rules_are_parsed	g_object_new
6	g_warning	gnome_desktop_thumbnail_factory_generate_thumbnail	gnome_desktop_thumbnail_script_exec	g_message
7	g_assert	gnome_get_language_from_locale	collect_locales_from_localebin	g_hash_table_lookup
8	g_hash_table_insert	gnome_get_country_from_locale	gnome_parse_locale	g_return_if_fail
9	Strlen	expand_thumbnailing_cmd	expand_thumbnailing_cmd	strcmp
10	g_print	gnome_parse_locale	gnome_desktop_thumbnail_factory_generate_thumbnail	g_str_equal

Table 6. Top 10 functions of emacs by their centralities.

Rank	In-Degree	Out-Degree	Betweenness	PageRank
1	Strlen	main	xmalloc	fprintf
2	DEFSYM	openp	lisp_to_value	xmalloc
3	Strcpy	initialize_environment	initialize_environment	exit
4	Defsubr	socket_connection	value_to_lisp	strcmp
5	MODULE_FUNCTION_BEGIN	scan_lisp_file	globals	strcpy
6	Xmalloc	decode_env_path	pop_getline	putc
7	value_to_lisp	init_cmdargs	fatal	strlen
8	Fprintf	member	dir_warning	defsubr
9	Staticpro	doprnt	doprnt	MODULE_FUNCTION_BEGIN
10	make_number	popmail	declaration	value_to_lisp

Table 7: Top 10 functions of git by their centralities.

Rank	In-Degree	Out-Degree	Betweenness	PageRank
1	die	cmd_main	get_oid_with_context_1	free
2	free	cmd_commit	get_oid_with_context	die
3	strbuf_release	cmd_clone	get_oid	strcmp
4	strcmp	pick_commits	get_oid_1	strbuf_release
5	error	cmd_tag	start_command	error
6	oid_to_hex	prepare_to_commit	run_command	strlen
7	fprintf	do_pick_commit	get_short_oid	oid_to_hex
8	strlen	start_command	unpack_trees	BUG
9	strbuf_addstr	cmd_receive_pack	diff_cache	strbuf_reset
10	strbuf_reset	cmd_fsck	run_diff_index	fprintf

Table 8. Top 10 functions of linux kernel by their centralities.

Rank	In-Degree	Out-Degree	Betweenness	PageRank
1	unlikely	SYSCALL_DEFINE2	console_unlock	kfree
2	mutex_unlock	copy_process	printk_deferred	might_sleep
3	mutex_lock	SYSCALL_DEFINE3	panic	mutex_unlock
4	kfree	SYSCALL_DEFINE5	vprintk_deferred	mutex_lock
5	u64	do_exit	vprintk_emit	unlikely
6	WARN_ON_ONCE	SYSCALL_DEFINE1	clockevents_program_event	u64
7	WARN_ON	load_module	clockevents_increase_min_delta	WARN_ON
8	this_cpu_ptr	SYSCALL_DEFINE4	clockevents_program_min_delta	WARN_ON_ONCE
9	BUG_ON	load_image_lzo	update_process_times	this_cpu_ptr
10	per_cpu_ptr	COMPAT_SYSCALL_DEFINE4	tick_periodic	BUG_ON

5 Conclusion and future directions

In this research, we analyzed the function-call graphs of four open-source software projects by using complex network analysis techniques, namely GNOME Library, GNU Emacs, Git, and Linux OS Kernel. We first downloaded the source code files of these software from their github repositories. Then, we generated their static call graphs in Graphviz dot graph format using pycflow2dot tool. Once the call graphs were available in this format, we used Cytoscape and Gephi to explore and analyze them both visually and topologically.

When we investigated the degree distributions of the graphs both visually and analitically, we observed Power Law distribution in all of them regardless of their sizes, meaning all the graphs were scale-free in compliance with the results of previous studies.

We also examined the existence of small-world property in the call graphs by comparing some topological features of the graphs with those of the corresponding random graphs that we algorithmically generated. Consequently, we showed that the software call graphs are small-world networks.

In a further analysis, we identified the most central (or important) functions in the call graphs by calculating in-degree, out-degree, betweenness, and PageRank centralities of the nodes. Although we did not have a comparison ground for assessing the relevance of these identified software units to the actual design decisions, they could still have critical importance for maintainability and reusability of the software systems.

For the analysis of communities embedded in the call graphs, we used the Modularity analysis method and observed that the call graphs exhibited a tendency to form communities that could indicate the modular and hierarchical design of the respective software systems. We showed that analysis of static function-call graphs of software systems through complex network analysis has the potential to reveal useful information about them. As a future direction, comparison of several software metrics collected via traditional software engineering approaches with the measures obtained via complex network analysis could reveal significant correlations.

6 References

[1] McConnell S. *Code Complete: A Practical Handbook of Software Construction.* 2nd ed. USA, Microsoft Press, 2004.

[2] Barabási AL, Pósfai M. *Network Science.* Cambridge, Cambridge University Press, 2016.

[3] Myers CR. "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs". *Physical Review E,* 68(4), 046116, 1-15, 2003.

[4] Yan K-K, Fang G, Bhardwaj N, Alexander RP, Gerstein M. "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,* 107(20), 9186-9191, 2010.

[5] Guo Y, Zhao Z-x, Wang W. "Complexity analysis of software based on function-call graph". *The 19th International Conference on Industrial Engineering and Engineering Management*, Changsha, China, 27-29 October 2012.

[6] Zhao D, Miao L, Zhang D. "Reusable function discovery by call-graph analysis". *Journal of Software Engineering and Applications,* 8, 184-191, 2015.

[7] Yu Qu, Guan X, Zheng Q, Liu T, Wang L, Hou Y, Yang Z. "Exploring community structure of software Call Graph and its applications in class cohesion measurement". *The Journal of Systems and Software,* 108, 193-210, 2015.

[8] Falci DHM, Gomes OA, Parreiras FS. "Complex networks analysis for software architecture: an hibernate call graph study". *ArXiv,* abs/1706.09859, 2017.

[9] GNOME Library. "GitHub-GNOME/Gnome-Desktop: Library with common API for Various GNOME modules". https://github.com/GNOME/gnome-desktop/ (01.07.2018).

[10] Gnu Emacs. “GitHub-Emacs-Mirror/Emacs: Mirror of GNU Emacs". https://github.com/emacs-mirror/emacs/ (01.07.2018).

[11] Git. "GitHub-git/git: Git Source Code Mirror". https://github.com/git/git (01.07.2018).

[12] Linux kernel. "GitHub-torvalds/linux: Linux Kernel Source Tree". https://github.com/torvalds/linux/ (01.07.2018).

[13] pycflow2dot. "pycflow2dot· PyPI". https://pypi.org/project/pycflow2dot/0.2/ (01.07.2018).

[14] GNU cflow. "GNU cflow". https://www.gnu.org/software/cflow/ (01.07.2018).

[15] Graphviz. "Graphviz-Graph Visualization Software". http://www.graphviz.org/ (01.07.2018).

[16] Zweig KA. *Network Analysis Literacy: A Practical Approach to the Analysis of Networks.* Austria, Springer-Verlag, 2016.

[17] Tunalı V. *Sosyal Ağ Analizine Giriş.* Ankara, Türkiye, Nobel Akademik Yayıncılık, 2016.

[18] Shannon P, Markiel A, Ozier O, Baliga NS, Wang JT, Ramage D, Amin N, Schwikowski B, Ideker T. "Cytoscape: A software environment for ıntegrated models of biomolecular ınteraction networks". *Genome Research,* 13(11), 2498-2504, 2003.

[19] Bastian M, Heymann S, Jacomy M. "Gephi: An open source software for exploring and manipulating networks". *International AAAI Conference on Weblogs and Social Media*, San Jose, California, USA, 17-20 May 2009.

[20] Hu Y. "Efficient, high-quality force-directed graph drawing". *Mathematica Journal,* 10(1), 37-71, 2005.

[21] Barabási AL, Albert R. "Emergence of Scaling in Random Networks". *Science,* 286(5439), 509-512, 1999.

[22] Watts DJ, Strogatz SH. "Collective dynamics of small-world networks". *Nature,* 393(6684), 440-442, 1998.

[23] Erdös P, Rényi A. "On Random Graphs". *Publicationes Mathematicae Debrecen,* 6, 290-297, 1959.

[24] Blondel VD, Guillaume JL, Lambiotte R, Lefebvre E. "Fast unfolding of communities in large networks". *Journal of Statistical Mechanics: Theory and Experiment,* 2008(10), P10008, 1-12, 2008.