A Case Study of Dependency Network for Building Packages: The Fedora Linux Distribution

Jiman Du[1], Jiaxin Zhu[2,3,4], Hui Li[2], Wei Chen[2,3,4], Lijie Xu[2,3,4], Jie Liu[2,3,4], Zhifeng Chen[5]
[1]School of Computer, Electronics and Information, Guangxi University, China
[2]State Key Lab of Computer Science at ISCAS, University of CAS, China
[3]University of Chinese Academy of Sciences,Nanjing, China
[4]Nanjing Institute of Software Technology, China
[5]MIIT Key Lab of Cloud Computing Standards and Applications, China Electronic Standardization Institute, China
Email: dujiman@st.gxu.edu.cn, {zhujiaxin, lihui2012, chenwei, xulijie, ljie}@otcaix.iscas.ac.cn, chenzf@cesi.cn

***Abstract*—To port the Linux distributions to a new Instruction Set Architecture (ISA), developers have to rebuild the software packages of the distributions. The complex dependencies of the software packages bring a great challenge. It is important to understand and properly handle the dependencies. We selected Fedora, a typical Linux distribution, and studied the dependencies within the software repositories of aarch64 and x86_64 architecture. We proposed a package dependency network framework to study the roles played by different packages. We obtained three network dependency patterns and proposed the corresponding division strategies which help developer build the source packages in parallel. Our study reveals that the key packages located at the root of multiple dependency chains significantly impact the division of the network, and their builds should be prioritized. Meanwhile, some packages with external dependencies can be temporarily masked to make a sub-network independent. Furthermore, the network dependency patterns are also observed in Fedora 33 riscv64 and OpenEuler riscv64. Our findings can help researchers have a better knowledge of Linux distribution dependency network and help practitioners conduct efficient package builds.**

Keywords—software repository, build dependency, software porting, dependency network

I. Introduction

With the advent of new ISAs such as ARM and RISC-V, developers of Linux distributions are seeking to port their distributions to these new architectures and establish corresponding software ecosystems [1]. However, porting a distribution to a new ISA is far from straightforward. Firstly, building software from source packages to binaries requires the support of other software known as build dependencies (reused software). The building process of a porting has to be started from scratch and conducted in dependency order. Secondly, the complex dependencies between source packages can be overwhelming for developers. Furthermore, porting a distribution to a new architecture involves rebuilding a large number of source packages, which is a time-consuming task.

Ye [2] focuses on the order of source packages building, using topological sorting to give the best building order. In this paper, we conducted an empirical study of the dependency relationship between source packages in Linux distribution software repositories. We crawled all packages, including source and binary packages, from 4 Fedora repositories for X86 (x86_64) and ARM (aarch64) architectures and 2 repositories of OpenEuler and Fedora 33 for the emerging RISC-V (riscv64) architecture. We built a package network for each repository based on the dependency relationships between source packages, and studied the networks for efficient porting.

We analyzed the connections of packages in the network to check whether the network can be divided into sub-networks for parallel building. Our results show that source packages can be aggregated into clusters, with sparser connections between the clusters than within the clusters. It implies that it is possible to fragment the network into sub-networks. We also obtained some insights of the role that different source packages play in the network. We found that the successful builds of the key packages belonging to multiple dependency chains are important for the builds of the dependent packages. These key packages have a significant impact on parallel building. Finally, according to the prior findings, we attempted to divide the dependency network into sub-networks and identified three dependency patterns and the associated division strategies.

The main contributions of this paper are as follows: (1) We built a package dependency network framework of Linux distributions; (2) We obtained three dependency patterns to help developers improve package build efficiency.

The remaining paper is organized as follows. Section 2 introduces our motivation and research questions. Section 3 presents the dataset. Section 4 illustrates our method and reports and the results. Section 5 introduces related work. Section 6 concludes.

II. Motivation and Research Questions

Porting a Linux distribution to new ISAs often requires rebuilding all of its software packages from scratch with unsupported dependencies. As depicted in Fig. 1, since the build of the source package *accountservice* [3] relies on the output of building *gobject-introspection*, the build of the former cannot be started until the successful build of the latter. This is a simple example of a dependency chain of source packages.

The numerous dependency chains form a large network, which may be split into smaller sub-networks for parallel builds. As shown in Fig. 2, after the successful builds of the *zlib*, *make*, and *gcc*, the network becomes two independent sub-networks which have 4 and 8 packages separately. They can be assigned to different teams for parallel builds.

To explore the division of the dependency network, we raise the following two research questions: **RQ1: What are the characteristics of the dependency network? RQ2: How can a dependency network be divided into independent sub-networks?**

This work is supported by the National Key R&D Program of China Grant 2021YFB1716000.
DOI reference number: 10.18293/SEKE2023-098

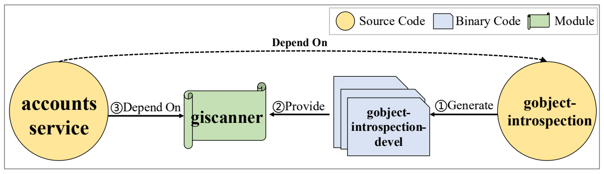


Figure 1. A dependency chain of two source packages.

III. Dataset

The dependency network of a repository can be defined as a directed network $G(V, E)$. $V = \{v_1, v_2, ..., v_N\}$ is the set of source packages in the repository, and $E = \{e_1, e_2, ..., e_M\}$ is the set of dependency edges, where $e_i = \{v_s, v_t\}$ $(i = 1, 2, ..., M)$. Source packages can be grouped into a cluster, i.e., a sub-network $G_{sub}(V_{sub}, E_{sub})$, where $V_{sub} \subseteq V$, $E_{sub} \subseteq E$. To examine the relationships between the sub-networks, some relevant external source packages are added into the original sub-network, which forms the merged sub-network.

To collect the edges between source packages in the dependency network, the following relations were extracted: the *generation* relation from an SRPM (source package) to an RPM (binary package), the *providing* relation from RPM to binary modules, and the *build dependency* relation from the SRPM to binary modules. These information can be acquired through the rpmspec [4] and rpm [5] tool. Take the dependency chain in Fig.1 as an example. Firstly, the rpmspec is used to parse the SPEC file of *gobject-instrospection* to extract the generating relation between the *gobject-introspection* and the *gobject-introspection-devel*. Next, the rpm is utilized to retrieve the list of modules provided by *gobject-introspection-devel*, which includes *giscanner*. Finally, the SPEC file of *accountsservice* is parsed again to obtain its build dependencies using rpmspec. In consequence, the relations of the source packages from *accountsservice* to *gobject-introspection* are obtained.

TABLE I. STUDIED DEPENDENCY NETWORKS

Architecture	Release Version	Number of nodes	Number of edges
aarch64	Fedora 34[a]	21494	155990
	Fedora 35[b]	22015	157897
x86_64	Fedora 34[c]	21620	157249
	Fedora 35[d]	22136	159215
riscv64	Fedora 33[e]	21949	144063
	OpenEule[f]	992	3483

a. https://mirrors.aliyun.com/fedora/releases/34/Everything/aarch64/.

b. https://mirrors.aliyun.com/fedora/releases/35/Everything/aarch64/.

c. https://mirrors.aliyun.com/fedora/releases/34/Everything/x86_64/.

d. https://mirrors.aliyun.com/fedora/releases/35/Everything/x86_64/.

e. http://fedora.riscv.rocks/repos-dist/rawhide/latest/riscv64/.

f. https://isrc.iscas.ac.cn/mirror/openeuler-sig-riscv/oe-RISCV-repo/riscv64/.

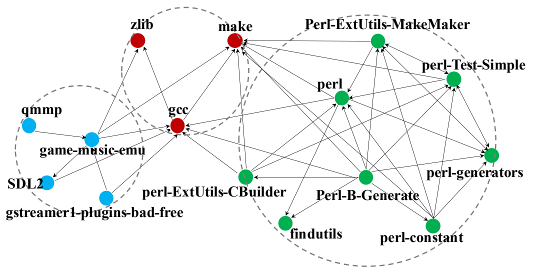


Figure 2. An example of network division.

To answer the two research questions, we crawled all packages in the 6 repositories mentioned in Section 1, the results of the network construction are presented in Table 1. Our initial analysis reveals that over 97% of the source packages are interconnected in the largest connected component. The remaining fragmented packages have minimal effect on network analysis. Thus, they are eliminated.

IV. Methodology & Results

A. Ans. to RQ1: Attributes of Source Packages Network

Dividing a network into sub-networks requires identification of community structures. The sparsity between communities can reduce the complexity of the division. Meanwhile, the efficiency of the building process may be impacted by the connections between the packages, as packages with a large number of dependencies may block the builds of many others. Therefore, we analyzed the internal structure and the degree distribution (dependencies distribution) of the dependency network.

Small-World: The average shortest path length (ASPL) [6] represents the efficiency of dependency transferred in the network, and the average clustering coefficient (ACC) [7] is the probability that packages gather into a cluster. We compare ASPL and ACC of Fedora dependency network (FedoraGraph) with a random directed network (RandomGraph) of the same scale. Taking Fedora 34 x86_64 as an example, as shown in Fig. 3, the FedoraGraph has a shorter ASPL (0.42) and a higher ACC (0.30), which indicates that the network has a small-world attribute [8], i.e., it is highly connected and aggregated. Nodes in the network tend to gather into clusters, and the relationships between the clusters are sparser than those within the clusters. This finding suggests that the dependency network can be divided into sub-networks.

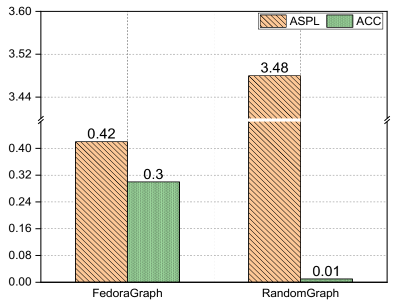


Figure 3. APSL and ACC of the FedoraGraph and RandomGraph.

Scale-Free: Fig. 4 shows the log-log plots of the degree distribution, where the horizontal axis represents the in/out degree of source packages and the vertical axis is the corresponding distribution. Our fitting results show that the degree distribution follows a power-law distribution (γ_{in} is 1.836, and γ_{out} is 2.099), indicating the scale-free attribute [9] of the network. The majority of the packages in the repository occupy upstream positions in the dependency chain and do not provide support for the builds of other packages. A large number of dependencies are concentrated in a minority of the packages, which we refer to as key packages. Consequently, the successful builds of key packages can significantly reduce the overall number of unsupported dependencies for the packages of a Linux distribution.

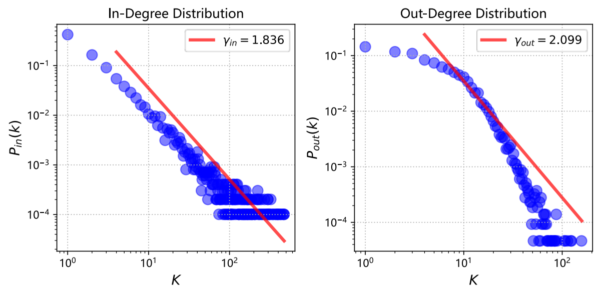


Figure 4. Degree distribution of the Fedora 34 x86_64 network.

B. *Ans. to RQ2: Dependency Patterns and Network Division*

We tried to divide the network through the following steps. First, we utilize the Louvain algorithm [10] for package grouping, as it shows good performance in large-scale networks. Then, we employed a network transformation approach, treating sub-networks as nodes, to simplify the relations for initial filtering. Finally, we divided the network and drew conclusions.

We define the average out-degree of a sub-network as shown in Eq. 1, $d_{out}(c, v)$ represents the out-degree of nodes within sub-network C to sub-network V, and n_c is the number of source packages in sub-network C.

$$d_{avg}(C, V) = d_{out}(C, V)/n_c \qquad (1)$$

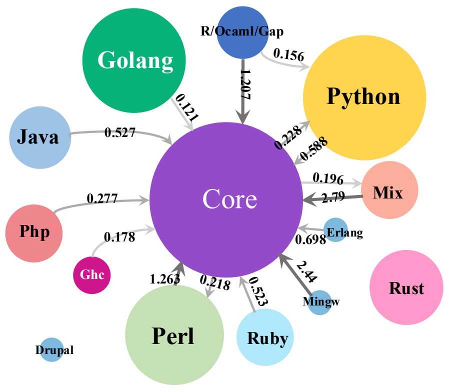


Figure 5. Dependency network at sub-network granularity.

To simplify the relationship between sub-networks, only the cases where $d_{avg}(C, V)$ is greater than 0.1 are considered. As shown in Fig. 5, the network is divided into 14 sub-network nodes, which still have the scale-free attribute. Except for the hybrid sub-networks (Core, R/Ocaml/Gap and Mix), the source packages in a sub-network are developed with the same programming language or framework. After the simplification, two isolated sub-networks (Rust and Drupal) emerge, which have the minimal dependencies on external and can be separated through masking a few dependencies. The dependencies of the other sub-networks are still too complex to separate.

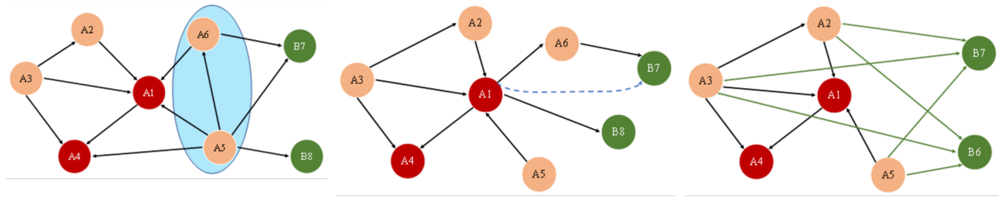
Based on the prior observations, we proposed two heuristics for network division: (1) Key packages in the sub-network must be made independent to ensure the builds of all related dependency chains. (2) The sub-network does not need to be completely independent of external dependencies. Developers can work on a sub-network when the majority of its packages can be built. With these heuristics, we identified three patterns of sub-network dependency as depicted in Fig. 6.

Partial dependency (PD): When the external dependencies are concentrated on the upstream packages (i.e., nodes with 0 in-degree), a sub-network has the partial dependency. In this pattern, although the sub-network still depends on packages from the external sub-network, the dependencies of most packages are within the sub-network itself. Developers can build this kind of sub-networks in advance. Fig. 6(a) depicts the merged sub-network A (A1-A6) to B (B7-B8). The red nodes (A1 and A4) are the key packages of sub-network A. A5 and A6 depend on B7 and B8 of sub-network B. In this pattern, the union of A5 and A6 does not support other source packages. Whether these nodes are successfully built or not has little effect on most of the source packages in the sub-network. Developers can put off the build of A5 and A6 when scheduling.

Core dependency (CD): The external dependencies of a sub-network are concentrated in the key packages, which are the foundation of multiple dependency chains and are critical to the builds of many source packages. The late builds of key packages will block all the other packages on the dependency chains. Developers must prioritize the builds of the key packages. As shown in Fig. 6(b), A1 and A4 are considered the key packages of sub-network A. A1 has two external dependencies: a direct dependency B8 and the transitive dependency B7 through A6, therefore, A1 is not able to be separated from the external, and the late build of A1 will block sub-network A. In this pattern, the key packages in the sub-network should have the highest priority.

General dependency (GD): The dependencies of most packages of a sub-network are a few external packages. The sub-network of this pattern cannot be easily separated from the external. As shown in Fig. 6(c), A2, A3, and A5 in sub-network A depend on B6 and B7 in sub-network B. Although the two key packages A1 and A4 do not form core dependency, the sub-network cannot be divided due to the dependency of B6 and B7. Developers should complete the builds of a few external source packages to make the sub-network only have partial dependency.

We also observe these patterns in the sub-networks of Fedora 33 riscv64 and OpenEuler riscv64. As demonstrated in Table 2, all the sub-networks, excluding the Mix sub-network, match one of the patterns and can be divided in the ways discussed above.



(a) Partial Dependency (PD) (b) Core Dependency (CD) (c) General Dependency (GD)

Figure 6. Dependency patterns of the sub-networks.

TABLE II. THE DIVISION RESULT OF SUB-NETWORKS IN FEDORA 33 RISCV64 AND OPENEULER RISCV64

Repository	Sub-Network	PD	CD	GD	Divisible
OpenEuler	Python	-	✓	-	✓
	Perl	-	✓	-	✓
Fedora 33	Drupal7	✓	-	-	✓
	Emacs	✓	-	-	✓
	Erlang	-	-	✓	✓
	Gap	-	✓	✓	✓
	Ghc	-	✓	-	✓
	Globus	-	✓	✓	✓
	Golang	✓	-	-	✓
	Hyphen	-	✓	-	✓
	Java	-	✓	-	✓
	Mix	-	✓	✓	-
	NodeJs	-	✓	-	✓
	Ocaml	-	✓	-	✓
	Perl	-	✓	-	✓
	PHP	-	✓	✓	✓
	Python	-	✓	-	✓
	R	-	✓	-	✓
	Ruby	-	✓	-	✓
	Rust	-	✓	-	✓

V. RELATED WORK

Software reuse often leads to dependency problems, and most of the studies focused on runtime dependencies. Several tools [11-14] have been proposed to address dependency conflicts and redundant dependencies in Python programs and Jar files. Li et al. and Prana et al. [15, 16] studied dependency conflicts and dependency vulnerability respectively. To our knowledge, only Ye [2] paid attention to the build dependency and provides a way to sort the build order of source packages of Linux distributions. Yao et al. [6] analyzed defects and changes of resilience during software evolution by modeling functions in the Android OS kernel. Similarly, Gou et al. and Gao et al. [17, 18] modeled and studied different software systems at the function level. Decan et al. [19] conducted an empirical study on seven software ecosystems and found that a few binary packages bear most of the dependencies, and the majority are unable to work without these dependencies.

VI. CONCLUSIONS

One of the biggest challenges faced by emerging ISAs is building the corresponding software ecosystem from scratch. For Linux distributions, developers must rebuild the software repository to support the emerging ISAs, where the build dependencies are very complex. In this paper, we conducted an empirical study of multiple repositories to explore whether the dependency network can be divided into sub-networks. Our findings reveal that the dependency network presents scale-free and small-world attributes. Source packages within the network tend to gather into clusters, which offers the potential to divide the network into sub-networks. The key packages in the network significantly impact the independence of the sub-networks, as they act as the root of multiple dependency chains and can block the build process of many other packages. We identified three patterns of sub-network dependency and provided the corresponding strategies of division. Our work can inspire further research on the dependency network and help developers efficiently conduct parallel builds of source packages.

REFERENCES

[1] "Linux Distro on RISC-V," https://riscv.or.jp/wp-content/uploads/Linux_Distros_on_RISC-V_Vietnam.pdf, Accessed: Apr. 24, 2023.

[2] A. D. Ye, "Research and Implementation of the compiling method for RPM with complicated dependency relationships," Master of Engineering, Institute of Computing Technology Chinese Academy of Sciences, Beijing, 2016.

[3] "Build Log for Package accountsservice," https://build.openeuler.org/package/live_build_log/openEuler:Mainline:RISC-V/accountsservice/advanced_riscv64/riscv64, Accessed: Apr. 24, 2023.

[4] "rpmspec," https://man7.org/linux/man-pages/man8/rpmspec.8.html, Accessed: Apr. 24, 2023.

[5] "rpm - RPM Package Manager," https://rpm-software-management.github.io/rpm/man/rpm.8.html, Accessed: Apr. 24, 2023.

[6] A. Yao, P. Sun, S. Yang and D. Li, "Evolution of Function-Call Network Reliability in Android Operating System," in IEEE Transactions on Circuits and Systems I: Regular Papers, 2020, vol. 67, no. 4, pp. 1264–1275.

[7] G. Fagiolo, "Clustering in complex directed networks," Physical Review E, 2007, vol. 76, no. 2, pp. 026107.

[8] D. Watts, S. Strogatz, "Collective dynamics of 'small-world' networks," Nature, 1998, vol. 393(6684), pp. 440-2.

[9] A. Barabási and R. Albert, "Emergence of Scaling in Random Networks," Science, 1999, vol. 286, pp. 509-512.

[10] V. D. Blondel, J. L. Guillaume, R. Lambiotte and E. Lefebvre, "Fast unfolding of communities in large networks," Journal of statistical mechanics: theory and experiment, 2008, vol. 2008, no. 10, pp. P10008.

[11] J. Wang, L. Li and A. Zeller, "Restoring Execution Environments of Jupyter Notebooks," 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, ES, 2021, pp. 1622-1633.

[12] E. Horton and C. Parnin, "DockerizeMe: Automatic Inference of Environment Dependencies for Python Code Snippets," 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 2019, pp. 328-338.

[13] H. Ye, W. Chen, W. Dou, G. Wu and J. Wei, "Knowledge-Based Environment Dependency Inference for Python Programs," 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, 2022, pp. 1245-1256.

[14] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A comprehensive study of bloated dependencies in the Maven ecosystem," Empir Software Eng, 2021, vol. 26(3), pp. 1-44.

[15] S. Li, J. Liu, S. Wang, H. X. Tian and D. Ye, "Survey of State-of-the-art Third-party libraries Conflict Dependency Problem," Journal of Software, doi: 10.13328/j.cnki.jos.006666.

[16] G. A. A. Prana, A. Sharma, L. K. Shar, D. Foo, Santosa, et al., "Out of sight, out of mind? How vulnerable dependencies affect open-source projects," Empirical Software Engineering, 2021, vol. 26(4), pp. 1-34.

[17] X. Gou, L. Fan, L. Zhao, Q. Shao, C. Bian, et al., "Multiscale Empirical Analysis of Software Network Evolution," 2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C), Hainan, China, 2021, pp. 1109-1118.

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

[19] A. Decan, T. Mens and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," Empirical Software Engineering, 2019, vol. 24(1), pp. 381-416.