Friedrich-Alexander-Universität Erlangen-Nürnberg

Faculty of Engineering, Department of Computer Science

ARMIN ROTH

MASTER THESIS

APPLYING INFORMATION DIFFUSION MODELS ON CODE REVIEW NETWORKS

Submitted on 8 June 2020

Supervisors:

Michael Dorner, M. Sc

Prof. Dr. Dirk Riehle, M.B.A.

Professorship for Open Source Software

Faculty of Engineering, Department of Computer Science

Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Erlangen, 8 June 2020

License

Erlangen, 8 June 2020

Abstract

Code review is one of the most important quality assurances of today's software development in open source as well as in industrial context. A side effect of reviewing code is the possibility of sharing knowledge between developers. While this is a common finding in software engineering research, until now, little has been done towards measuring the spread of knowledge. We therefore apply methods of information diffusion, well known in other research disciplines, to social networks constructed of code review data. In this thesis we give an introduction to information diffusion in code review, propose a measurement model for analyzing the information flow during code review and evaluate the model using case study research on two large and mature open source systems. We especially show differences in constructing the underlying code review networks regarding the factor time and the resulting consequences when applying diffusion models. We discuss the applicability of our model with regards to its flaws, possible improvements and differences in types of networks and diffusion models suitable for measurement.

Contents

1 Introduction

1.1 Original Thesis Goals

The original goal of this thesis was to conduct exploratory data analysis on code review data of the Linux kernel and the FreeBSD project. Therefore data of both projects should be gathered and analyzed using methods of social diffusion.

1.2 Changes to Thesis Goals

Since no literature has been published on analyzing code review data using contagion models or other techniques of social diffusion as this thesis was written, we decided to focus on initially showing the practical applicability of such methods on code review data. We therefore used the gathered data to conduct an evaluative case study to show the possibilities of analyzing code review data through applying contagion models on code review networks.

2 Research

2.1 Introduction

Code review is one of the most important quality-control mechanism in the open source community. (Rigby, German, Cowen, & Storey, 2014) Needless to say, it also plays a fundamental role in proprietary software development and major technology companies have modernized their code review processes tremendously in the recent past. (Bacchelli & Bird, 2013)

Although the main intended purpose of code review is the reduction of defects in software engineering, a non negligible side effect can be found in the sharing of knowledge in a team or community developing software. (Bacchelli & Bird, 2013) One could even argue further that code review itself is a process based on the transmission of information through a social network. In the following we consider an active exchange of information through code review as knowledge sharing.

While this seems reasonable, almost no work has been done in measuring the spread of information inside those communities.

In software engineering research, the analysis of social networks created from communication data generated during development, has evolved into an effective method to gain useful insights into development communities. Nevertheless, only analyzing properties inherent to the structure of a network can prevent one from gaining a more detailed insight into the information spreading capabilities of a social network, especially in regards to a time factor. We therefore propose to use techniques of information diffusion to analyze social networks created on code review data.

Adapted from Rogers (2003), Information Diffusion describes a process by which information spreads through a social network over time. The diffusion of innovation, social diffusion, knowledge or information diffusion, in general, are all similar terms describing how various entities can spread throughout a network. While widely used in other disciplines, little attention has been payed to applying methods of information diffusion in software engineering research.

Therefore, we present a model to apply models of information diffusion on code review data in order to analyze the spread of knowledge in a network of developers. To evaluate if the application of information diffusion is a useful means to investigate code review data we conduct a case study, analyzing code review networks of the Linux kernel and the FreeBSD project as two matured, important and influential open source projects.

2.1.1 Research Question

The goal of this thesis is to take an initial step in understanding how information can spread between developers when doing code reviews. It is unknown which results can be gathered when applying information diffusion models to social code review networks. Therefore, our research question can be summarized as follows:

- How can the spread of information in code review networks be measured using models of information diffusion?

2.1.2 Contributions

This thesis is one of the first to approach applying information diffusion models on social networks generated by code review data. Therefore, we contribute the following in this thesis:

- We give an introduction to the field of information diffusion and how it can be applied to code review networks.
- We present a measurement model, which can be used to estimate the spread of information in code review networks.
- We conduct a case study of information diffusion in two code review networks, based on two of the most important open source operating system projects, to evaluate if the suggested methods yield meaningful and expressive results.

2.1.3 Outline

The further structure of this thesis will be as follows: Section 2.2 gives a short overview of related work, especially of what has been done in analyzing code review using social networks as a foundation, the origins of the theory of diffusion and where to find more detailed insights about it. In Section 2.3 fundamental concepts of this thesis are explained. What is code review, what are social networks and how can they be constructed from code review data? Further, an introduction to the field of information diffusion is given. Section 2.4 describes our model to measure information diffusion in code review networks in detail.

Section 2.5 shows our procedure for evaluating the proposed measurement model for information diffusion in code review networks using case study research. Section 2.6 contains the result of this case study. In Section 2.7 the results, our proceeding in the case study and the presented measurement method are further discussed. Finally we give a conclusion and an outlook to future work in Section 2.8.

2.2 Related Work

The related work of this thesis can be split in multiple topics. First we will cover previous work on applying social network analysis on code review data, since code review networks are our primarily used data structure. Furthermore, we will show work, discussing the spread of knowledge during code review. At last, we will give a short overview of the origins and applications of information diffusion in other disciplines.

The importance of code review in software engineering is well known and the analysis of its practices and processes, strengths and weaknesses has become an active research field of its own. While Social Network Analysis (SNA) is a technique broadly applied in software engineering research to analyze general communication data during software development as shown by Amirfallah, Trautsch, Grabowski, and Herbold (2019), only little attention has been explicitly paid to SNA based on code review data aside from more traditional qualitative and quantitative analysis.

Preliminary work was conducted by Yang et al. (2012) and Hamasaki et al. (2013) investigating the possibilities of using social network analysis (SNA) in code review research. In a case study, code review contributor roles and activities of the Android Open Source Project were analyzed in relationship to the network structure. They further analyzed those relationships in other open source project and investigated how the most important contributors can be found using SNA and how the position of a reviewer in the network relates to the qualtiy of code reviews. (Yang, 2014; Yang, Yoshida, Kula, & Iida, 2016)

Furthermore, Bosu and Carver (2014) used SNA to differentiate developers into core and peripheral groups and moreover investigated how the reputation of a developer, determined this way, affects the time to receive feedback on a code review request.

Bacchelli and Bird (2013) as well as Baum, Liskin, Niklas, and Schneider (2016b) have analyzed modern code review in open source and industry context regarding the motivation behind its usage. A key result of both works is the motivation of additional benefits generated by reviewing practices besides finding defects in code. One of those benefits lies in the spreading of knowledge.

Rigby and Bird (2013) compared the review processes of a variety of different open source and industry projects regarding their differences and similarities. They found mutual characteristics shared by all projects, indicating general principles of code review practice. Furthermore, they attempted to measure knowledge sharing during code review by measuring the number of files a developer knows about and has contributed to or reviewed code on.

In comparison, we attempt a much more general approach in measuring the spread of knowledge. We rather measure the underlying information spreading capabilities of a network. To the best of our knowledge, until now no research has been conducted on analyzing information diffusion in social networks generated by code review data or other means of software engineering.

The theory of information diffusion itself however is well known and has long been applied to plenty of different research disciplines, but only in recent years gained importance in technical domains. Amongst other originated in the field of epidemiology and biology to analyze the spread of diseases (Anderson & May, 1979) and the field of sociology to study collective behavior (Granovetter, 1978), it was soon realized that those techniques can be used in a more generalized way (Goffman & Newill, 1964). In technical domains the concepts can for example be used to analyze the spread of information via phone calls (Herrera, Armelini, & Salvaj, 2015), tweets (Cha, Haddadi, Benevenuto, & Gummadi, 2010) or emails (Liben-Nowell & Kleinberg, 2008) to name only a few.

Zhang, Mishra, Thai, Wu, and Wang (2014) as well as Shakarian, Bhatnagar, Aleali, Shaabani, and Guo (2015) give an introduction over the most important diffusion models used in various disciplines together with their properties and possible problems.

2.3 Research Context

In the following section the general context of this thesis will be depicted closer, to understand the concept of information diffusion in code review networks.

2.3.1 Code Review

Code review, sometimes also called peer review, is a software quality assurance activity where one or multiple human, so called "reviewers", check software by viewing and reading source code after its implementation. At least one person performing the review must not be the original author of the code. (Baum, Liskin, Niklas, & Schneider, 2016a)

While the main goals of code review are finding defects and improving general code quality, other intended side effects include the transfer of knowledge – the reviewer and author gain knowledge about coding best practices, new ways to solve problems or other technical understandings –, a sense of mutual responsibility on a team – the ownership of the code is collective – and finding better technical solutions when working together. (Bacchelli & Bird, 2013; Baum et al., 2016b)

Historically code review is a formal, highly structured group review, typically done line by line in multiple phases as described by Fagan (1976), called *code inspection*. Today most teams in industry adopted more lightweight *modern code review* practices already present in open source projects, which can be described as an informal, continuous and regular, tool- and change-based code review. (Bacchelli & Bird, 2013; Baum et al., 2016b; Rigby & Bird, 2013)

Reviewing code, especially using modern code review, requires a huge amount of constant, continuous communication and can be seen as a social interaction between humans. Therefore, social networks can be created from the arising communication data.

2.3.2 Social, Temporal and Code Review Networks

Social Networks

"*Social networks* are networks in which the vertices are people, or sometimes groups of people, and the edges represent some form of social interaction between them, such as friendship." (Newman, 2018) The vertices are often referenced as actors and the edges as ties. Commonly, social networks are modeled as a mathematical graph with actors as its nodes and exchanged messages, or general communication relationship as edges.

Temporal Networks

While social networks can and have been modeled using static graphs they usually evolve over time. When modeling communication through static (often weighted) networks, the time dimension inherent to the communication data gets lost. (Holme & Saramäki, 2012)

Holme and Saramäki (2012) and Rossetti and Cazabet (2018) therefore sum up different types of Temporal Networks to incorporate time into a network:

- *Snapshot networks.* The evolution over time is represented as an ordered series of static networks. For each relevant point in time a network snapshot is captured.

- *Temporal networks.* Each edge in the networks is either associated with a timestamp or a time interval, which indicates the activity of a edge. Those networks are also referred as *contact sequences* when a timestamp is used and as *interval graphs* when using time intervals.

Snapshot and temporal networks are also called *dynamic networks* and are a representation of multilayer and multiplex networks as described by Newman (2018).

Furthermore, it should be noted that increasing time resolution goes along with increased graph complexity and in consequence computational power needed to perform graph analysis. (Rossetti & Cazabet, 2018)

Code Review Networks

A *code review network* is a social network consisting of developers as actors and their communication during code reviews as ties. The actors of a network should belong to the same community that develops software together.

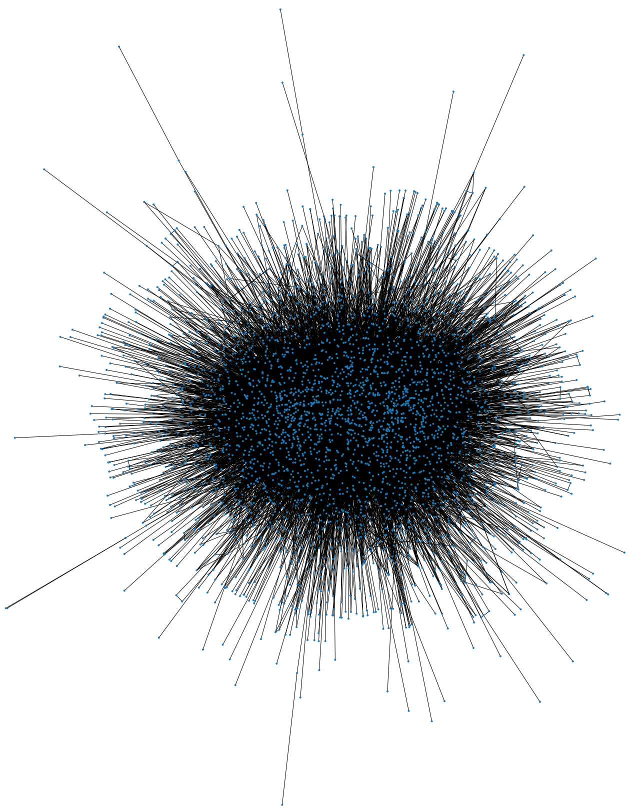


Figure 2.1: Code review network constructed by data of the LKML in 2018

The network graph may be modeled directed or undirected, depending whether it is desired to differentiate between code submitter and reviewer or not. The edges can be weighted or contain attributes if different types of message should be handled unalike, as well as the nodes can have attributes, to divide developers by contributors and reviewer or other means. Furthermore, code review networks often evolve over time.

Figure 2.1 shows an exemplary code review network created on code review data of the Linux Kernel Mailing List (LKML) during the year 2018. This figure shows the amount of communication data arising during code review in large software projects as well as the possible complexity of social networks created on this data. Visual and classical static SNA approaches are therefore unsatisfactory, which is why new modeling approaches are needed.

2.3.3 Information Diffusion

Social diffusion, innovation, knowledge, or information diffusion are all similar terms describing the same process of a variety of spreading entities in a (social) network. Diseases can spread in a network of actual people, information, knowledge and innovations can spread in communication networks based on various mediums like social online networks, email, telephony and others, even the spread of computer viruses can be analyzed by the same means. For this thesis, we settled with the term information diffusion, since information is one of the broadest concept of entities which can spread in a social network.

Various approaches have arisen to analyze information diffusion. One way to categorize today's techniques is to divide them into topological-statistical and dynamic modeling approaches.

Since the structure of a network naturally influences its capabilities to diffuse information, studying network topology gives valuable insights into how information can spread. For example, the presence and distribution of strong and weak ties can influence the spreading capabilities of a network depending on the contagion process as shown by Granovetter (1983) or Centola and Macy (2007). In short, weak ties, referring to lose relationships between individuals often bridge long distances in a social network, thus being able to promote contagion processes. Another research area falling into this category is the detection of popular topics. For example, by utilizing term frequencies the outbursts of popular topics can be detected or predicted in social networks. (Guille, Hacid, Favre, & Zighed, 2013)

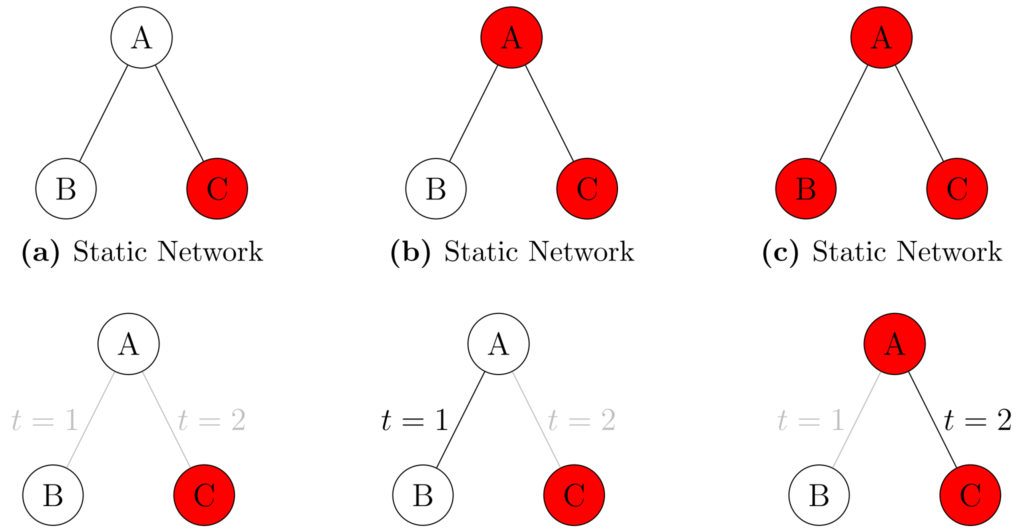
The second option is to model how information spreads through a network. This approach can further be devided into opinion and contagion models. (Rossetti et al., 2018)

Contagion models or epidemic models simulate the spreading of items in an disease like manner. Agents typically are modeled to have a simple state like susceptible, infected or recovered, and change their state through different update rules over time. Models of *simple contagion* are influenced by parameters like the contagiousness or the length of the infectious period of the disease. (Rossetti et al., 2018) The most popular representative for simple contagion models is the family of SI models (SI/SIS/SIR and others) based on the work of Kermack and McKendrick (1927). *Complex contagion* models by contrast take into account that the probability of contagion can depend on the number of exposures. (Min & San Miguel, 2018) Therefore, they allow to model phenomena like peer pressure or similar. Popular representatives for complex contagion models can be found in the independent cascade model (Kempe, Kleinberg, & Tardos, 2003) or threshold based models derived from Granovetter (1978).

Opinion Dynamic Models try to show how different opinions or topics are spreading through a network. In general similar to epidemic models, an agent can not only have discrete, but moreover continuous states representing the degree to which they has adopted an opinion. Also different competing opinions can be modeled using a multitude of states. Furthermore, external influence can be modeled by using a static individual, which is connected with all nodes in the network, overcoming the closed world assumption which underlies most epidemic models. (Rossetti et al., 2018)

Besides the attempt to further refine and unify contagion models to better approximate reality as can be seen at Milli, Rossetti, Pedreschi, and Giannotti (2018a) or Min and San Miguel (2018), recent research tries to capture dynamic processes using dynamic networks as a foundation for applying diffusion models or topological network analysis. (Holme & Saramäki, 2012; Zhan, Hanjalic, & Wang, 2019)

In various issues considered by SNA the factor time does not matter necessarily, however this is not the case for diffusive processes. In Figure 2.2 a small example on how information diffusion processes can be distorted when using different networks as modeling foundation, can be seen. Figure 2.2a - 2.2c show a diffusion process in a static network, Figure 2.2d - 2.2f in a temporal network – more precise, a contact sequence. Both networks have the same starting condition: Node C is informed. After two iterative steps, the whole network received the information in case of the static network. When looking at the temporal network, node B and A only have contact at t=1. Because A does only know of the information dispensed by node C since t=2, but not at t=1, node B will never receive this information. This underlines the transitive properties of information diffusion.



(a) Static Network **(b)** Static Network **(c)** Static Network

(d) Temporal Network **(e)** Temporal Network **(f)** Temporal Network

Figure 2.2: Information spreading in static vs. temporal networks

2.4 Measuring Information Diffusion in Code Review Networks

When looking at how code review is performed from a network point of view, typically only local parts of few people in a sub community participate in a concrete message exchange. However, what we are interested in is not how a review of a specific patch takes place, but rather how a generic piece of information can spread through the network during these short periods of communication.

In the case of communication during code review, various types of information can diffuse through the whole network. For example, changes in APIs or architecture, the introduction of new tooling software, programming guidelines and best practices, security issues and even more.

2.4.1 Measurement Approach

We propose to measure the possible information flow during code review by applying contagion models on code review networks. Looking at a hypothetical single piece of information spreading through a network, we want to know the following:

- Estimate a lower time limit until the information spread stalls
- Estimate the percentage of maximum reached nodes
- Gain insights into the spreading dynamics

Given a social network based on code review data the following properties influence the outcomes of our measurement variables besides the networks' capabilities itself:

- The type of information spreading
- The set of nodes initially coming up with new information
- Individual adoption properties of a node

2.4.2 Process Description

To measure information diffusion in code review networks, at least the following steps are necessary:

- *Collecting the code review data.* Most modern code review tools like Gerrit or Phabricator provide interfaces for retrieving data to support automation or data analysis.
- *Preparing the data.* Depending on the available data it can be desired to filter it further by time or project topics. For each submitted patch and subsequent comments the contributor should be extracted as well as the communication data itself. This includes at least the timestamp of each message to build a relationship between the different contributors. Depending on the used data source it might be necessary to unify multiple entities of the same developer if it is possible for him to have multiple accounts or to unify data models of different data sources.
- *Creating a social network.* Using the prepared data a static or dynamic social network must be created. We propose suited possibilities in Subsection 2.4.3.
- *Applying a contagion model.* Finally a suitable contagion model can be configured and applied on the created network. Two exemplified models are described in Subsection 2.4.4

2.4.3 Network Construction

Various possibilities exist to create social networks from code review data. We propose the following methods for the analysis of information flow during code review:

For each Developer who has written a patch or commented on an patch a node is placed in the graph. For each message exchanged during the discussion of a patch a vertex between all developers actively participating in the discussion is added. Afterwards all edges between two developers are unified and the number of connections is added as the weight of the remaining edges. Thus, the created multigraph is converted into an equivalent weighted graph. This way a static graph, incorporating the communication frequency, can be produced.

To construct a snapshot graph, this procedure can be executed multiple times on varying consecutive time frames of desired length.

For the creation of a temporal graph in form of a contact sequence, for each message exchanged during the review of one patch, an edge between the author of the message and all recipients contributing to the discussion is created with the timestamp of the message as attribute. Furthermore, an interval graph can be generated by using the timestamp of patch creation as a start date and the timestamp of the last posted comment as end date and defining an edge with this time data between all participants of a patch discussion. This can be compared to a communication window opening and closing.

Those methods can be seen as the basic possibilities to create code review networks. Many additional options exist to further refine the structure of a network. Besides using patches and their subsequent comments to generate a network, additional status information like patch acceptance or rejection can be considered as relevant communication data when creating code review networks. Additionally, one could further differentiate the nodes in terms of contributor and reviewer or model the communication flow using directed graphs.

2.4.4 Model Selection

When looking at a hypothetical single most important piece of information, which is relevant for each node, the most fitting model is a simple network flooding algorithm where each node submits the information to all its neighbors immediately after it has received it. Depending only on the network structure and the set of inital spreaders, this way a lower bound for the needed number of hops to spread information through a network can be derived.

Nevertheless, contagion models allow us to take further variables into account. We will describe the models we initially applied in our following case study. Since this was the first attempt to apply dynamic diffusion models to code review networks and the outcomes were unknown, we decided to apply simple and well known models. The selection of the SI model as a model of simple contagion and the threshold model as a model of complex contagion therefore seemed suited and can be seen as two possible models to measure information diffusion.

Overall, the SI model and its akin are the simplest possible approaches to model diffusion dynamics. It was first introduced mathematical by Kermack and McKendrick (1927) to model the spread of disease. While the model was original described using differential equation we use a slightly adopted version operating algorithmically on a graph.

Each member of the network can have either the state S (Susceptible) or I (Infected). At the beginning each node is susceptible with the exception of a set of initial infected nodes. The model is executed in an iterative simulation. In each iteration step an infected node contaminates its neighbor with an infection probability. This model assumes that an once infected person stays contagious for the rest of the simulation. Common adoptions of this model are therefore adding a recovery possibility, with which each node changes back into the susceptible state S (SIS model) or if it is wanted to model a disease with immunity, an extra recovered state R (SIR modeled) can be introduced.

We selected the SI model over enhanced forms of it because it makes the fewest assumption over the disease or items spreading. When setting the infection probability to 1, it assembles a simple flooding algorithm. In the case of this model, we are mainly interested in the throughput capabilities of a network, since each node will be infected after a finite time frame if the network is fully connected. However, it is possible that a node receives an information but is not able to turn it into knowledge for its own good, since it is too complex or the node is simply not interested in it. Since multiple studies have shown that models of complex contagion are better suited for modeling the spread of information in social networks (Min & San Miguel, 2018) we also selected a complex model.

The threshold algorithm used in this thesis was first described by Granovetter (1978). Whether a node becomes infect during a pandemic depends on the infection state of its neighbors. The more neighbors are infected the more likely the node itself will become infected. Hereby, each node can have an individual threshold of how many infected neighbors are needed to infect itself. Therefore, an individual percentual threshold is assigned to each node. The model works in iterative steps. In every iteration each node is inspected. If the percentage of infected neighbors are above its threshold the node becomes infected, otherwise nothing happens.

The SI model as well as the threshold model have been adopted for dynamic networks among others by Milli et al. (2018a, 2018b). In short, for each iteration at time t the infected nodes and edges of $t-1$ are evaluated regarding their influence on the graph at the current time.

The selected models are expected to be a rough approximation of the real diffusion processes because of their simplicity. While more sophisticated models of information diffusion exist, this is still an active area of research and for now out of scope in the context of this thesis. Nevertheless, possible solution approaches are noted in Section 2.7.

2.5 Research Approach: Evaluating Case Study

To evaluate if dynamic models of information diffusion can be used to measure the spreading capacities of code review networks we apply them in an evaluative case study according to Yin (2017). The main goal of this case study is to show the viability of our proposed method and analyze differences resulting of the underlying network model. Furthermore, we will evaluate if the model simulation with the selected models can achieve meaningful results.

2.5.1 Case Selection

The Linux Kernel and the FreeBSD projects are the selected units of analysis. We choose open source software projects since their code review data is public available. Both cases are of utter importance in the field of (open source) operating systems and are furthermore among the oldest and largest open source projects in terms of code base size and number of active developers. While the two projects are open source operating systems they differ in their code review practices, general development processes and the project governance.

In the following we give a short overview of the key aspects of the projects regarding history, project governance, development and code review processes.

Linux

The development of the Linux kernel was started in 1991 by Linus Torvalds and soon emerged to be one of the most important foundations of modern open source operating systems. Today Linux is used in various applications and forms one of the biggest open source project with 13000 to 14000 changesets, contributed by about 1800 developers each release cycle of two to three month. (Corbet, 2019) To develop a project of this size, a unique process has established itself.

The kernel is logically divided into a hierarchical structure of multiple so called subsystems. This can for instance be a subsystem for networking, specific architecture support, memory management or similar. Each actively developed subsystem has a responsible maintainer and most of the time is managed in its own version of the kernel source tree. (kernel development community, n.d.-a)

The development process relies heavily on the use of mailing lists. If a developer writes a patch they wants to be included in the Linux kernel, they has to post it on the relevant mailing lists of subsystems affected by the patch. Typically larger patch are devided into smaller parts and send as a so called patchset. Then everybody involved in those subsystems can comment on the code changes. In an iterative process the patch author can now post new versions of the patch, incorporating feedback. Once the maintainer is content, they accepts the patch and integrates it in his repository. It can be noted that this does not have to be the whole original patch, but also just cherry picked parts. Alternatively the patch can be rejected. Linus Torvalds then incorporates patches preselected by his top level subsystem maintainers into the mainline kernel repository. (kernel development community, n.d.-a)

Since a few years, most mailing lists are tracked by an instance of the Patchwork patch tracking system. Patchwork acts as a wrapper around mailing lists. It filters out relevant emails and provides a web interface organizing patches and comments, their version and other useful information. Maintainers can set a status to each patch, e. g. under review, accepted or rejected and gain a clear overview over all patches submitted to their subsystem. (Patchwork Developers, 2019) Therefore, Patchwork represents a good interface for gathering code review data of the Linux kernel.

BSD

Originating from the Berkeley Software Distribution developed since 1977, FreeBSD was started as its own project in 1993 and is one of the most important open source operating systems today, providing the base for various well known applications and services, among them for example Sony's recent PlayStation models or Netflix's OpenConnect appliances delivering over 30% of all internet traffic of North America. (FreeBSD Documentation Project, 2020) Currently it is developed by around 400 permanent active developers.

Active developers can be granted write access to the central SVN repository. They are called committers. Furthermore, almost 2400 people have contributed to FreeBSD on a occasional base to the project. Every two years the committer elect a so called core team, which guides the general direction the project is taking and has further organizational functions like the nomination of new committers. (FreeBSD Documentation Project, 2020)

The main repository is divided in 3 parts:

- *src* the kernel and important userland programs
- *ports* third party software packages, independent of kernel and userland ported to FreeBSD
- *doc* the FreeBSD manual and manpages shipped with the system

In comparison to Linux, FreeBSD forms a complete operating system, including essential userland programs, and not only a kernel. Thus in its whole, it is probably more comparable to a Linux distribution like Debian, Fedora or Arch. However, it should not be a problem to compare the two projects regarding their information transmission capabilities.

While FreeBSD still discusses general development topics on mailing lists, the community switched to using the tool suit Phabricator in recent years for reviewing submitted patches of new features. (FreeBSD Documentation Project, 2019) Phabricator is an open source tool suit developed by Phacility.inc facilitating developing software collaborative. Besides the classical *review then commit*, FreeBSD also provides the ability for *commit then review* through Phabricator. Nevertheless, developers are encouraged to always ask for feedback before committing their code.

Furthermore, FreeBSD also maintains a Bugzilla instance. Apart from the the possibility to report bugs, new contributors without commit rights are advised to post their patches on this site. When doing so, the keyword `[PATCH]` should be written in the synopsis of the report. (Hubbard, Lawrance, & Linimon, 2020) These are then reviewed by experienced FreeBSD developers and may be integrated into the system. Depending on the decision of the responsible developer, the patch may also be discussed further again on Phabricator before integration.

2.5.2 Data Gathering and Preparation

For conducting the case study, we gathered the code review data from both projects and prepared them for further analysis.

Data retrieval

Linux The development of the Linux kernel has been studied thoroughly in the past. Most previous research used the mailing lists stored as mbox or pubin files as base for their analysis. This brings up the necessity of preprocessing - for example parsing the email headers for information like the sender, recipient, the send time and so on. Furthermore, when analyzing code review, it may be necessary to filter out irrelevant emails not related to a patch, but rather containing general discussion. Xu and Zhou (2018) realized the opportunity of using Patchwork as a data source, as this provides all relevant email data already preprocessed and enhanced with additional meta-information when Patchwork is actively used by a subsystem community. While it was previously needed to manually analyze the HTML pages of each patch, Patchwork version 2.0 introduced a REST API for a more simple automatized retrieval of patch data. (Patchwork Developers, 2018)

We assume that the subsystems using Patchwork are representing all major parts of the Linux kernel and thus we should get a representative view of the Linux kernel development.

As a first step, all subsystem using Patchwork had to be identified. Most subsystems mailing lists relevant for the Linux kernel are indexed on "https://patchwork.kernel.org/" (kernel development community, n.d.-a). Nevertheless, some patchwork instances exist at other places.

Therefore, the MAINTAINERS[1] file of the kernel was parsed and all relevant mailing lists with existing Patchwork instances were extracted. For this case study we retrieved the last revision published on Github in 2018.

Each maintainer of a subsystem is listed in this file with additional information like his email address, files and directories of the subsystem, source code management (SCM) tree location, the mailing list relevant to the area or an available Patchwork project. In total there were 1791 unique entries for existing subsystems, of which around 150 are currently unmaintained or obsolete. 243 unique mailing lists and 45 unique Patchwork projects were linked in the MAINTAINERS file. In total 6 domains exist hosting Patchwork projects: "patchwork.freedesktop.org", "patchwork.kernel.org", "patchwork.linux-mips.org", "patchwork.linuxtv.org", "patchwork.open-mesh.org", "patchwork.ozlabs.org". The LKML is indexed as a single project at an own Patchwork instance reachable under "https://lore.kernel.org/patchwork/"

All projects available on those domains were parsed. Each Patchwork project description features an unique identifier referencing the project and one referring the the mailing list it is indexing. For each mailing list referenced at a Patchwork project it was checked if an entry in the MAINTAINERS file exists. If so, all patches were downloaded using the REST API. In total data of 92 mailing lists was retrieved.

Currently, multiple Patchwork versions are in use. Most organizations use Patchwork Version 2.0 as for the time this thesis was written. Freedesktop.org choose to develop its own fork in the past (freedesktop.org, 2019). Since the API is restricted, i. e. does not provide access to relevant information, the code review data of subsystems developed on freedesktop.org could not be included in the analysis.

BSD The codereview data of FreeBSD can be centrally accesses on its Phabricator instance reachable under "https://reviews.freebsd.org/". Phabricator provides a JSON-RPC based API to automate querying, creating and editing of its objects. (Phacility, Inc., n.d.) Thus, all data could be easily retrieved. The data contains all revisions and their subsequent comments and additional metadata.

FreeBSD's Bugzilla instance is reachable under 'https://bugs.freebsd.org/bugzilla/' and provide a REST API for automated data retrieval. All bugs containing `[PATCH]` in their synopsis which were posted or last edited in the specified time frame were collected together with their comments.

[1]https://github.com/torvalds/linux/blob/master/MAINTAINERS

Timeframe While initially collecting data of a broader time span we decided to use a 5 year span from 01.01.2014 to 31.12.2018 as a basis for the further data preparation and analysis, since Phabricator is only in use since 2014 and while the first linux mailing list dates back to 1998 this amount of data is difficult to handle and, furthermore, mostly only of historical interest.

Data Preparation

Linux When working with mailing lists, as a first step emails unrelated to code review should be filtered since not only code review but also general discussions take place on a development mailing list. Therefore, it is assumed that Patchwork processes the mailing lists correctly and fetches all patches, answers and status changes ("acked by", "reviewed by", etc.), if an email is formatted according to the actual guidelines (kernel development community, n.d.-b).

A common problem when analyzing mailing lists is the practice of the same person writing with multiple different email addresses over longer time spans or even simultaneously. In consequence, different email addresses of one person should be clustered to get more accurate results in later analysis.

Patchwork creates a new "people"-entry for each email address participating in the mailing list and maps a name based on the "From Tag" of the email header. E. g. for the tag `From: "Joe Doe" <jd@example.org>` the entry `Joe Doe | jd@example.org` is created in the Patchwork database. While it is possible for users who created a Patchwork account to add email aliases, it seems that this is not a widely adopted practice on most patchwork instances for now.

Consequently, the user entries should be matched to get accurate results. The general matching approach was taken form Bird, Gourley, Devanbu, Gertz, and Swaminathan (2006), but slightly adapted in parts as described in the following part. While Bird et al. generated big cluster containing many false positives, which they manual broke down afterwards, it was tried to get the best possible result without having to manually process the data.

- Normalize Name
  As a first step, the name is normalized mostly similar to Bird et al. All strings are converted to lowercase only, suffixes in parenthesis or after hyphens are removed (e. g. joe doe - company inc.), generic terms are stripped from the name (e. g. admin, linux, arm, kernel, dev, etc.). The name is split in first and last name (and possibly middle names) based on white spaces and commas. All remaining punctuation is removed, the string is encoded in ASCII removing or if possible converting non ASCII symbols. (e. g. é $\rightarrow$ e) and multiple whitespaces are merged to one.

  Those normalized names are than clustered using a custom defined distance based on the following:

- Name Similarity
  This step was done the same way as by Bird et al. Names are compared using the Levenshtein distance, which measures the minimum steps inserting, deleting or replacing letters to get from one word to another. For example *Joe Doe* is quite similar to *Jane Doe*, but completely different from *Max Mustermann.* The comparison is done independent of first and last name, thus *Joe Doe* is equal to *Doe Joe.* Naturally, a smaller distance results in a higher similarity.

- Names-Email Similarity
  Based on the Levenshtein distance it is compared if the email contains the first and last name. Furthermore, if an email address starts with the letter of the first name and continuous with the last name or vice versa, this is considered a match by Bird et al. Here, this rule is only used if the tuple to compare with has no entry for a name, since this rule led to many false positives. Thus, *Joe Doe* matches *jdoe* or *djoe*, but only if no name is present in the last tuple (this is the case if no name entry existed in the email header).

- Email Similarity
  As a last matter the email addresses are compared. If the emails are completely equal it is considered a match. Otherwise the local parts of the emails are compared using the Levenstein distance again. We further extended this step compared to Bird et al. If the local part is only a generic term (e. g. mail, dev, linux, etc.) the domain part excluding the top level domain part is used as comparison (e. g. mail@joedoe.com). Also if the local part only consists of a natural name [2] or is less than three characters long, no similarity can be granted to prevent to much false positives. E. g. joe@a.com, joe@b.com is not considered a match.

- Cumulative Similarity
  The highest similarity of the above is used as the final value of a tuple comparison.

Of all downloaded code review data (patches and comments) of the Linux project 21004 (name, email)-pairs were initially extracted before the cleanup. This could be reduced to 20553 (name, email)-tuples after an initial name normalization.

The clustering was done using an agglomerative approach based on the pairwise distance of all tuples. The clustering was executed multiple times varying the used distance threshold to optimize the resulting clusters.

Figure 2.3 shows how the distance threshold impacts the number of resulting clusters. After manually inspecting the results, it was decided to set the distance threshold to 5. This leads to a rather strict name matching, with almost no false positives. For 11101 (name, email) pairs no match was found. 2664 cluster consisted of 2 entries, 743 of 3 and 414 of 4 or more. Out of all clusters with more than 3 (name, email) pairs, only 1 single wrongly matched pair could be identified, leading to a false positive rate of 0,2% for the 412 biggest cluster.

[2]The names corpus of M. Kantrowitz and B. Ross is used for natural name detection. http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/nlp/corpora/names/

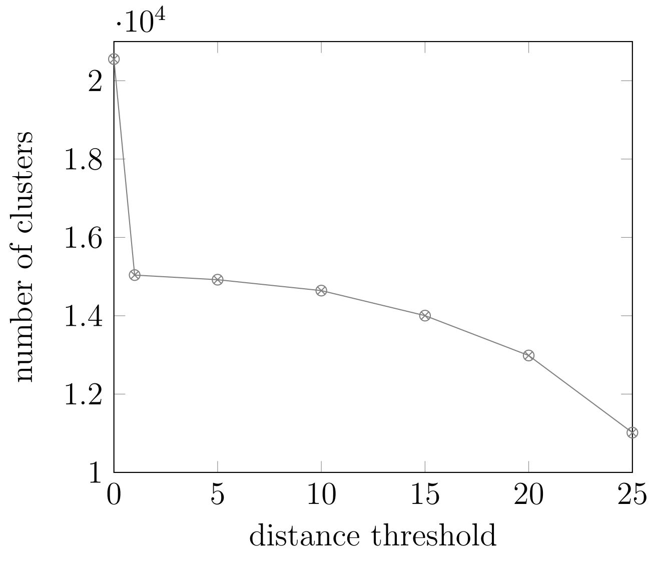


Figure 2.3: Name clustering - cluster sizes

BSD In Contrast to Linux, BSD development is much more centralized. Normally committers should only have one Phabricator account, nevertheless some users seem to have created multiple accounts over the time, for unknown reasons.

Since Bugzilla is used as an additional data source besides Phabricator, accounts of both platforms were linked if possible. Therefore, a similar approach to the Linux name matching was chosen. Bugzilla accounts are retrieved as "name", "real name" -pairs, whereas the name in all cases at FreeBSDs Bugzilla instance are email addresses. Phabricator similarly returns "username", "real name" -pairs, but contrary to Bugzilla not all usernames necessarily are email addresses.

As a first step, the usernames of both platforms were normalized. Afterwards, a Phabricator account was matched to an Bugzilla account based on usernames and email adresses if possible.

Out of 3116 Bugzilla accounts 885 could be matched to Phabricator accounts.

Summary

In total 1832627 Linux patches and 2028758 subsequent comments were collected. Therefrom 540733 patches were posted on the LKML and 1291894 on other lists.

Regarding Free BSD 23536 revisions and 80343 attached comments could be collected. Therefrom, 1068 revision were found in the doc tree, 7166 in the port tree and 13392 in the src tree. Moreover 74 audits were done using Phabricator in the selected time span. On top of that, 8512 problem reports containing a patch and 51772 attached comments were found on Bugzilla 2023 related to the base system, 6274 regarding ports & packages and 215 concerning documentation.

2.5.3 Used Tools

We decided to use *Python* (v. 3.7) as the foundation for conducting the case study, since it offers great support and high flexibility when conducting data analysis. The data preparation and analysis was mainly done by using interactive *Project Jupyter* notebooks (Project Jupyter, 2020) as development environment. The most important libraries used were *pandas* (pandas development team, 2020) and *NumPy* (NumPy developers, 2020) for data handling and basic statistical computations. *scikit-learn* (Pedregosa et al., 2011) and *fuzzywuzzy* (SeatGeek, 2020) were used for clustering the user accounts. The social networks were created and examined using *NetworkX* (NetworkX developers, 2020), respectively *DyNetX* (Rossetti, 2017) – an adaption of the former – when working with dynamic networks. Finally *NDlib* (Rossetti et al., 2018) was selected for evaluating the possible use of diffusion models.

We selected NDlib primarily due to the broad range of available models, the possibility to use temporal networks, its included visualization framework, the possibility to easily define further models and the fact it is in continuous active development. Nevertheless, it should be noted that other libraries exist to model diffusive dynamics on complex networks. Amongst are for example *Epigrass* or *Nepidemix* building up on Python or *EpiModel*, a package for R. A more detailed list can be found by Rossetti et al. (2018)

2.6 Research Result

In the following section we will present the results of applying different diffusion models with varying parameters on code review networks. While we attempt to model the flow of information, we will also use the terminology of infections and diseases for the following passage for easier readability. Information therefore equals disease and the infection of a node is equivalent to a node noticing and processing a piece of information.

2.6.1 Model Application on Weighted Networks

SI-Model

To begin with, we decided to apply the SI model on a static weighted network. The application of this model implies that the information spreading is relevant to each node in the network. On a static, fully connected graph, after a finite set of iterations, all nodes will have adopted the information.

In a first attempt, all patches and comments initially posted in 2018 were selected as the base for the creation of the code review networks. The static weighted network was created as described in Section 2.4.3. The resulting BSD network has 643 nodes and 4027 edges. The Linux network has 5627 nodes and 49490 edges. Furthermore, we only selected the biggest connected component by removing disconnected clusters only consisting of few nodes occurring in Linux. The resulting graph contains 5611 nodes and 49482 edges.

One percent of randomly selected nodes were set as initial infected. This equals a small informed minority with information new and unknown to the rest of the network. The infection probability β was set to 1. This means a node will always infect another node they has contact with. Thus our spreading information is of theoretical single most importance since it will reach every node, and every node will transmit it immediately. The SI model was executed 100 times for the BSD network and 500 times for the Linux network. The number of executions was based on the number of nodes in each network. For each execution we selected a varying random set of initial infected nodes.

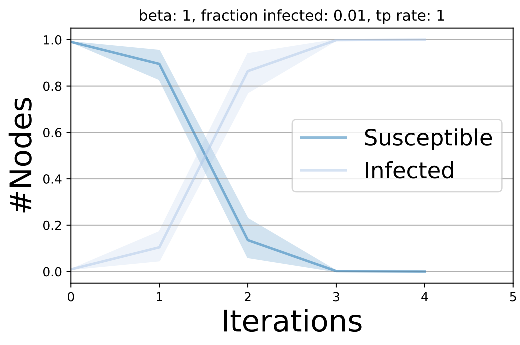


Figure 2.4: Diffusion Trend BSD - SI model (2018)

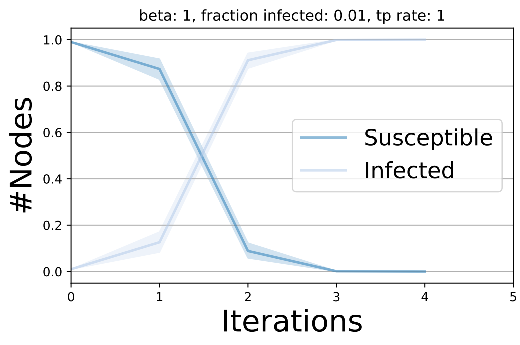


Figure 2.5: Diffusion Trend Linux - SI model (2018)

Figure 2.4 and 2.5 show the average diffusion trend of the executed model. The average percentage of susceptible and infected nodes of the whole population is plotted for each iteration, together with its inter percentile range of 10% to 90% in form of translucent areas.

Both networks show very similar behavior in terms of infection spread. On average after 3.8 iterations all nodes of the network are infected in case of BSD and after 4 iterations in case of Linux. However, in the BSD network 98% and in the Linux network 99% of all nodes already become infected after 3 iterations on average. The spread is rising moderately in the first and third iteration, while the second iteration has the highest slope, assembling a sigmoid curve. A noticeable difference is that the infection curve of the BSD networks shows a slightly higher variance in the beginning and the final rise of infections.

In the following we will visualize the results primarily by plotting the infected nodes over time. Nevertheless, other common visualizations exist and should be noted. One approach often used to visualize diffusion processes is to plot the incidence, the number of new occurring cases, for each iteration as shown in Figure 2.6 and Figure 2.7

A further alternative for visualization lies in plotting the infection spread in the network. In Figure 2.32 and Figure 2.33 in Appendix A, the networks of two model executions with a random set of initial infected nodes are drawn using the Fruchterman-Reingold force-directed algorithm (Fruchterman & Reingold, 1991) to lay out the nodes. The algorithm tends to place nodes with high centrality more towards the center of the

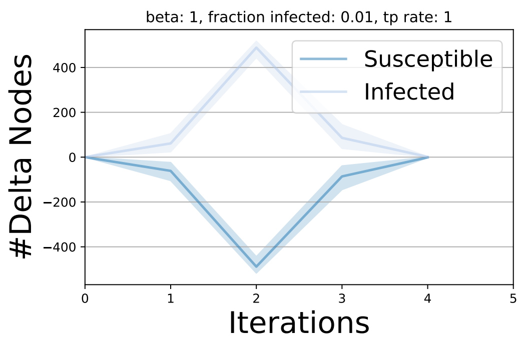


Figure 2.6: Incidence BSD - SI model (2018)

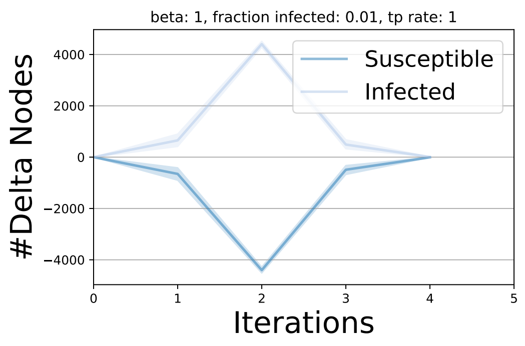


Figure 2.7: Incidence Linux - SI model (2018)

drawing. The color of each nodes shows in which iteration the node became infected. Nevertheless, this approach is hardly overseeable and since the plotted graphs in general are dimension or else scale less it is hard to draw conclusions from this depiction.

The two parameter adjustable at the SI model are the infection probability β and the set of initial infected nodes. Varying β has a similar effect on both networks as can be seen in Figure 2.8 and 2.9. When leaving the percentage of initial infected nodes at 1%, on average the time needed to reach the whole network was slowed down from 3.8 iterations to 6 iterations for $\beta = 0.75$, 8.5 iterations for $\beta = 0.5$ and over 10 iterations for $\beta = 0.25$ when looking at the BSD network and from 4 iterations to 7.5 iterations for $\beta = 0.75$, to 9 for $\beta = 0.5$ and over 10 iterations for $\beta = 0.25$ when looking at the Linux network.

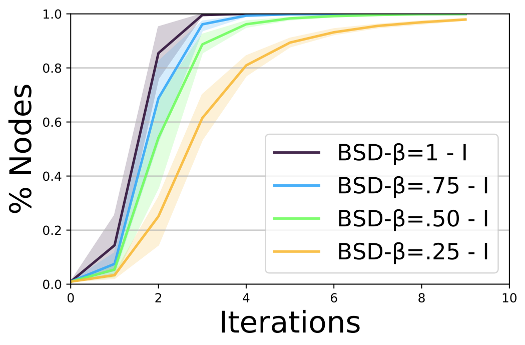


Figure 2.8: Varying β BSD - SI model (2018)

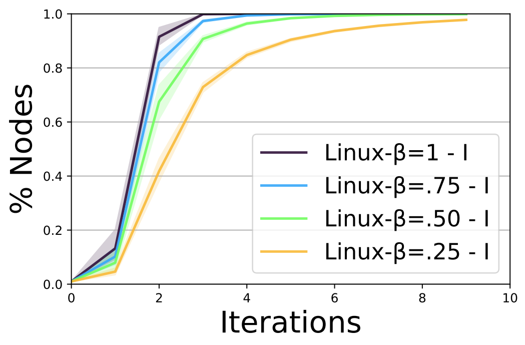


Figure 2.9: Varying β Linux - SI model (2018)

Figure 2.10 and 2.11 show the effects of varying the percentage of initial infected nodes, while leaving β at 1. Increasing the initial infection percentage to 10% speeds up the process. After the second iteration 98% of all network nodes are infected in both cases, the last 2% of nodes need another iteration in case of BSD and 1.7 in case of Linux to become infected. Decreasing it to 0.1% [3] results again in a total of 4 needed iterations, but in comparison to 1% infected nodes, the biggest rise appears around one iteration delayed. Furthermore, it can be noted that in general the smaller the number of initial infected nodes, the more it matters which nodes are selected.

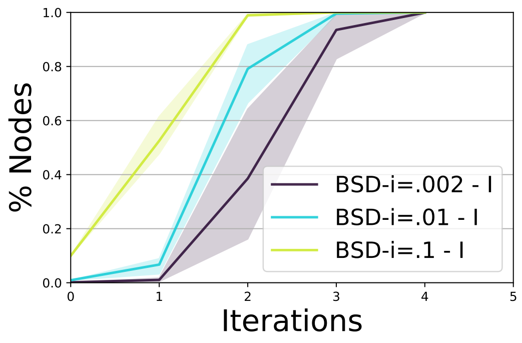


Figure 2.10: Varying initial infection percentage BSD - SI model (2018)

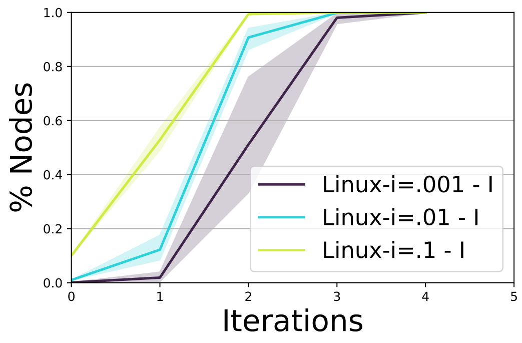


Figure 2.11: Varying initial infection percentage Linux - SI model (2018)

Therefore, we decided to sort the developers of our networks based on how interconnected they are. We selected the one percent developers with the highest node degree and the one percent of developers with the lowest node degree as initial infected set. Selecting the most networked nodes as initial infected nodes for applying the SI model with $\beta = 1$ can be seen as the maximum possible throughput of our networks when starting with 1% infected nodes.

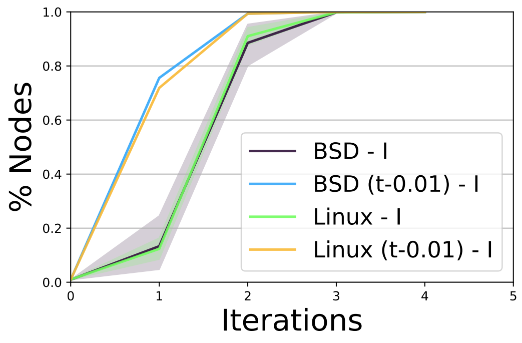


Figure 2.12: Varying initial infected nodes (top) - SI model (2018)

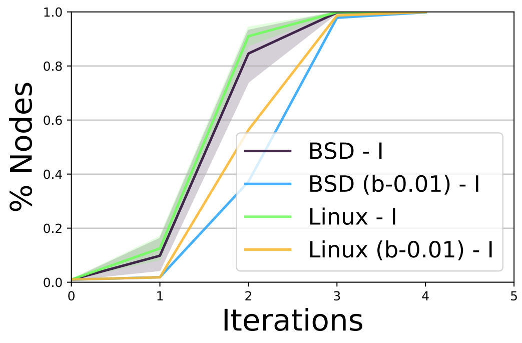


Figure 2.13: Varying initial infected nodes (bottom) - SI model (2018)

The outcomes, in comparison to the curves with randomly selected initial infection sets, can be seen in Figure 2.12 and 2.13. *t* references the selection of nodes with the highest node degree, *b* the selection of nodes with the lowest. Both networks behave highly

[3] The value was rounded up in case of BSD to have at least one infected node

similar. When electing the top 1%, after the first iteration over 70% of all nodes are infected in both cases. After the third iteration all nodes are infected in case of BSD; in case of Linux the process still needs 4 iterations to finish completely, even though 99.3% of all nodes are infected after the second iteration. When selecting the bottom 1% of nodes based on their degree the contagion processes finishes again after the 4th iteration for BSD, respectively the 5th iteration for Linux.

Threshold-Model

Since information are not diseases and studies have shown that information is spreading rather through complex than simple contagion processes (Min & San Miguel, 2018), we decided to further apply a threshold model. In this model each node has an individual threshold to adapt new information. If the percentage of neighbors adapting the information exceeds the threshold, the node itself will adapt the information. The threshold was set as the ratio of messages sent and received by a node in comparison to the communication traffic of the whole network. Therefore, the weighted degree of each node was divided by the sum of all the weights of the vertices of the whole network. This results in a mean threshold of 0.0031 with a standard deviation of 0.0080 for BSD, and a mean threshold of 0.0004 with 0.0014 standard deviation for the Linux kernel.

Figure 2.14 and Figure 2.15 show the average trend lines of susceptible and infected nodes for the applied threshold model, together with its inter-percentile range. The models were once more executed 100 respective 500 times with varying random initial infected sets of 1% of all network nodes.

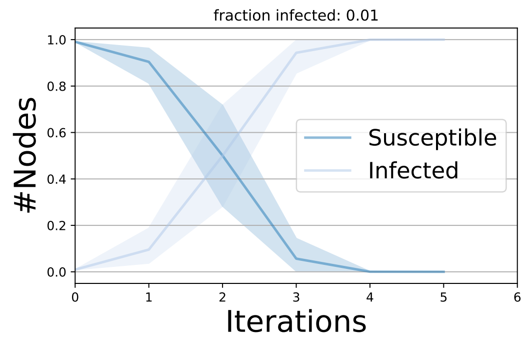


Figure 2.14: Diffusion Trend BSD - Threshold model (2018)

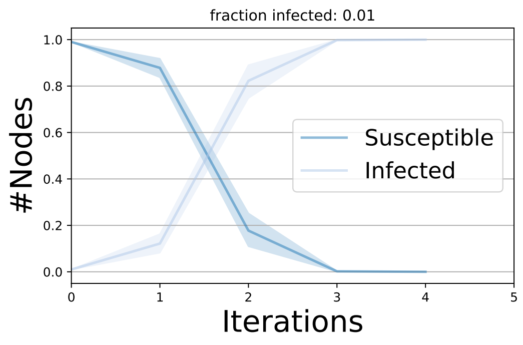


Figure 2.15: Diffusion Trend Linux - Threshold model (2018)

Again, both networks show very similar behavior in terms of infection spread. The curve shapes look highly similar to the previously run SI-model. Like before, after 4 iterations on average all nodes of both networks are infected in all executed simulations. With this threshold model, the infection curve of the BSD network depends even more on the initial set of infected nodes, since the possible outcome of infected nodes per iteration varies much more than for the Linux simulation.

As previously, we run the threshold model varying the percentage of initial infected

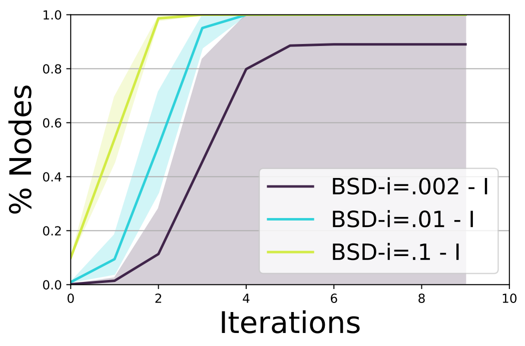


Figure 2.16: Varying the initial infection percentage BSD - Threshold model (2018)

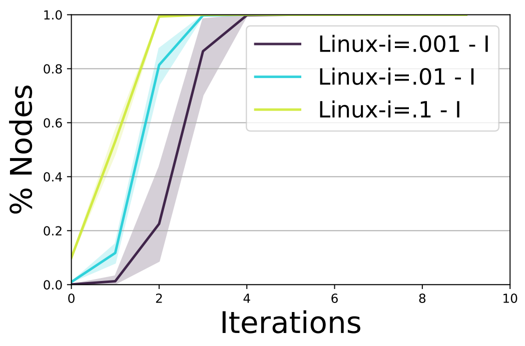


Figure 2.17: Varying the initial infection percentage Linux - Threshold model (2018)

nodes. Figure 2.16 and 2.17 show the resulting contagion processes. Increasing the initial infection percentage to 10% results in similar curves as when modeled with the SI model. In both contagion processes the large majority of nodes are infected after the first two iterations. The Contagion process of BSD is finished again after 3 iterations, Linux after 3.6 on average. Decreasing it to 0.1% slows the process down by 1 iteration for Linux almost as previously; in the case of BSD by contrast the contagion process on average stalls after 4.5 iterations only resulting in 88% infected nodes on average. This happens because 0.2% of infected nodes only equals one actual node, making it possible to stop the information flow right at the beginning when the threshold of its neighbors are to high.

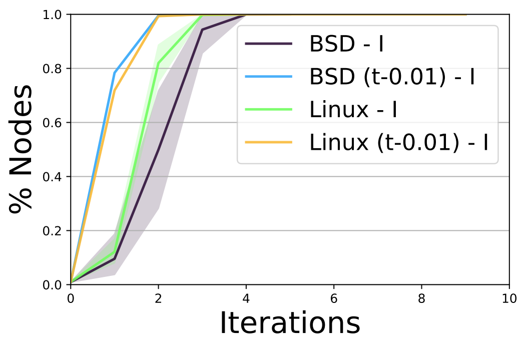


Figure 2.18: Diffusion Trend Comparison - Top 0.01 infected nodes - Threshold model (2018)

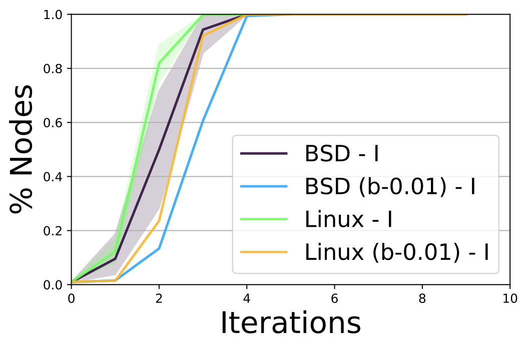


Figure 2.19: Diffusion Trend Comparison - Bottom 0.01 infected nodes - Threshold model (2018)

At last, we executed the model again, selecting the most and least interconnected nodes based on their degree as initial infection sets. Figure 2.18 shows the resulting trend curves when selecting the nodes with the highest degree, Figure 2.19 when selecting the nodes with the lowest degree as initial infection set. As a reference the graphs with randomly selected initial infection sets are plotted for comparison. As with the

SI model, in the second iteration already over 70% of all nodes are infected in both networks. Like previous, after the third iteration all nodes are infected for BSD and after the fourth all for Linux.

When selecting the nodes with the lowest degree, the curves look more similar to the one with randomly drawn nodes, nevertheless the iterations needed to infect the whole network is shifted towards 5 iterations for both networks and the iteration with the highest infection rise is shifted from the second to the third iteration. However, the large majority of nodes is already infected after 4 iterations.

Timeframe Based on Code Review Length

Since possible new ideas, opinions and similar can spread rather fast when simulated on static graphs, the question arose if the selected time slice of one year may be to broad. To select a more reasonable time frame the average discussion time of a patch was calculated. Therefore, the discussion time of each patch was set as the difference between the creation date of the patch and the last comment posted.

In the year 2018 the mean discussion time for a patch was roughly 9 days for a patch submitted to a Linux related mailing list. For all BSD patches submitted via Phabricator in 2018 the mean discussion time of 40 days was calculated. For BSD patches submitted via Bugzilla an average discussion time of 29 day was determined, leading to an average discussion time of 39 day for BSD overall.

In consequence for the Linux kernel time slices of 1 week and for FreeBSD time slices of 1 month were used to create multiple weighted graphs. On average the code review network of BSD has 148 nodes for each month, with a standard deviation of 20 and the code review network of Linux 900 nodes per week with a standard deviation of 77. BSD contains 31, Linux 60 nodes being part of each network sliced by the average code review length.

For each network the threshold model was executed 10 respective 50 times with varying random initial infection sets. The average node threshold calculated on the time sliced networks was 0.0025 and 0.0002. Additionally, each network was executed another time with the top and bottom 1% interconnected nodes as initial infected nodes. Figure 2.20 and 2.21 shows the month of May and calendar week 20 as representative curves for all time slices. A comparison plot of all curves is plotted in Appendix B Figure 2.34 and Figure 2.35.

Both projects are relatively stable over all time slices. On average the model stagnates at 90% infection rate after 5 iterations for BSD and on 93% after 6.3 iterations for Linux. For Linux the last week of the year could be identified as outlier, only reaches slightly above 80% infection rate. A closer look reveals that only one day was part of the last week in 2018. For BSD the total span of infected nodes of multiple month reaches from around 5% to 95%. This can be explained through the structure inherent to the networks. The network graphs sometimes consists of multiple separated subgraphs, where one contains the majority of nodes, whereas the other only contain a handful of nodes. When 1% of the graphs population is drawn randomly, it may happen

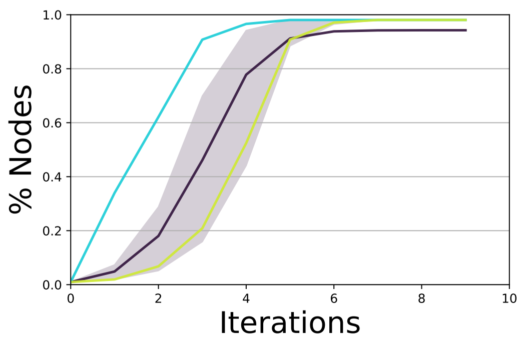


Figure 2.20: Diffusion Trend BSD (May, 2018)

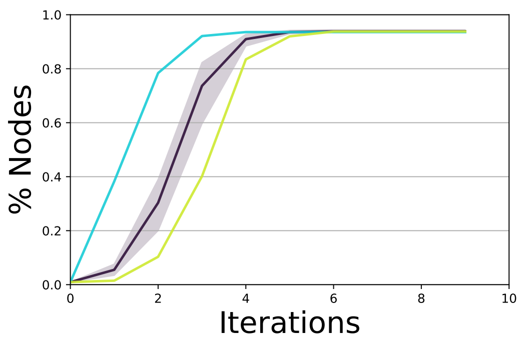


Figure 2.21: Diffusion Trend Linux (Week 20, 2018)

that the only 2 infected nodes are drawn from the smallest of multiple subgraphs, and therefore possibly remain the only infected nodes. When selecting the top or bottom 1% interconnected nodes 96% and 65% nodes are reached on average after 4.6, respectively 5.1 iterations for BSD and 94% after 5, respectively 7 iterations for Linux.

Conclusions

Using diffusion models on static weighted networks provides a good possibility to compare the general transmission capabilities of different networks. We could show that information can spread mostly similar in the communities of BSD and Linux. Using SI and threshold models, with the given parameters set, the large majority of all nodes will receive the information spreading in both networks in under 5 generic iterations. When comparing the spreading process in the BSD and Linux community, one can register that in the case of BSD the progression of information spreading, depends a lot more on the nodes originating with the information to spread.

When selecting multiple time slices of one year, it can be shown that the capabilities of information transmission stay stable over time. Reducing the analyzed time on the average code review length reduces the total number of nodes receiving the information when applying a threshold model. Furthermore, the process gets slowed down compared to the time slice of one whole year.

2.6.2 Model Application on Dynamic Networks

Snapshot Based

The previously generated networks created on the average communication duration during a code review, provide a good possibility to examine the model execution on snapshot based dynamic networks. We used the previously generated weighted networks spanning over time frames of average code review duration to create a snapshot network for the year 2018 for both projects.

We executed the SI model as well as the threshold model on both networks. Again the selection of initial infected nodes was done at random and based on their degree. The threshold values calculated on the static networks of the whole year were used for each node. In the following, each iteration refers to one month respectively one week.

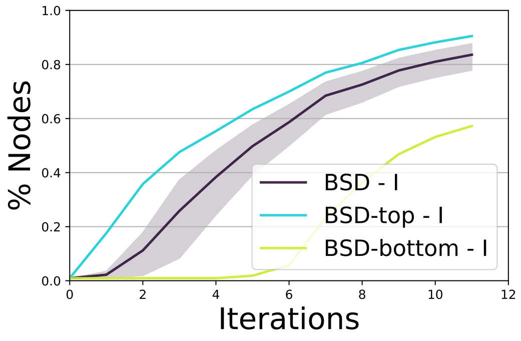


Figure 2.22: Diffusion Trend BSD SI model - snapshot based (2018)

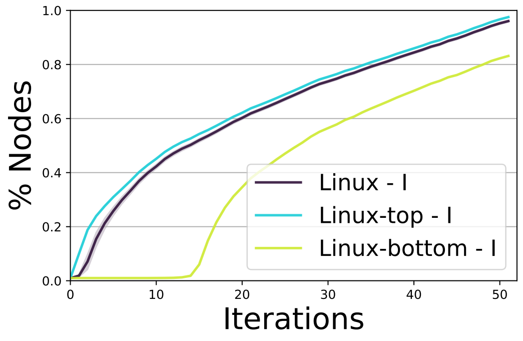


Figure 2.23: Diffusion Trend Linux SI model - snapshot based(2018)

Figure 2.22 and 2.23 show the development of infections when selecting 1% infected nodes at random or based on their degree at the beginning. On average 84% nodes are infected at the last iteration at BSD and 96% at Linux. When selecting the nodes with the highest or lowest degree, the networks reach an infection rate of 90% and 57% for BSD, respectively 97% and 83% for Linux. The late infection outbreak when selecting the bottom nodes can be explained by the fact that this is the point of their first participation in a code review.

When selecting the nodes with the highest degree, 50% of the BSD network is infected after 4 iterations - corresponding to a time frame of 4 month, when looking at Linux 50% are infected after the 13 iterations - corresponding to a time frame of 3 month [4].

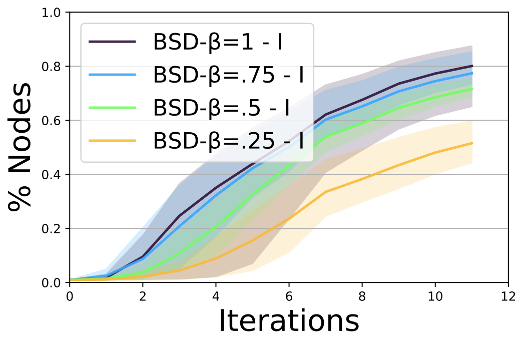


Figure 2.24: Varying β BSD SI model - snapshot based (2018)

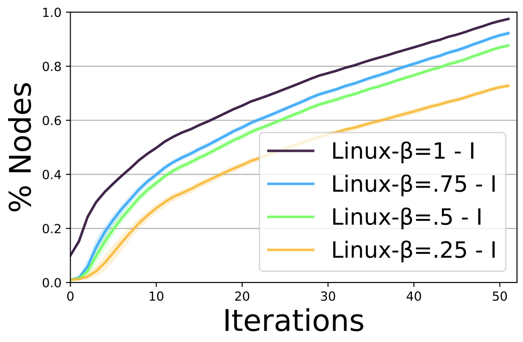


Figure 2.25: Varying β Linux SI model - snapshot based(2018)

[4] $1\ month = 4.345 \times week$

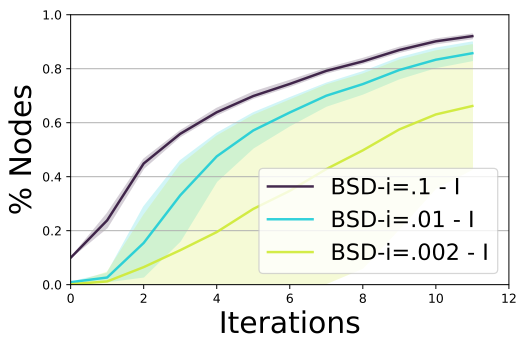


Figure 2.26: Varying the initial infection percentage BSD
SI model - snapshot based (2018)

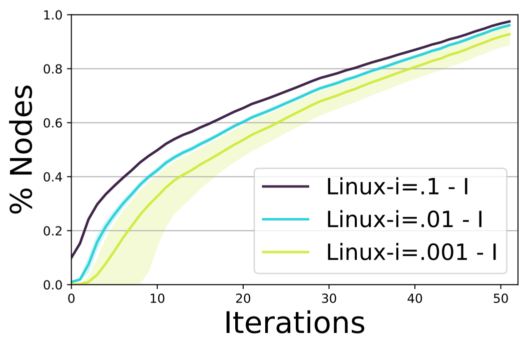


Figure 2.27: Varying the initial infection percentage Linux
SI model - snapshot based(2018)

Figure 2.24 and figure 2.25 show the effect of varying β in a snapshot based network. When varying β in BSD, the average infection rate drops from 84% for $\beta = 1$ over 79% for $\beta = 0.75$ and 70% for $\beta = 0.5$ to 50% for $\beta = 0.25$. In contrast for Linux the average infection rate only drops from 96% for $\beta = 1$ over 92% for $\beta = 0.75$ and 87% for $\beta = 0.5$ to 72% for $\beta = 0.25$. Varying the percentage with which one node infects another, seems to have an higher impact in the BSD community compared to the Linux community.

Figure 2.26 and Figure 2.27 show the effects of varying the percentage of initial infected nodes. Selecting 0.1%, 1% or 10% initial nodes increases the final infection rate from 64% over 84% to 91% for BSD and from 92% over 96% to 98% for Linux. 50% of the network is reached after 4, 6 and 10 iterations (4, 6 and 10 month) for BSD and 11, 14 and 18 iterations (2.5, 3.2 and 4.1 months) for Linux. The plot shows again that the outcomes in the BSD network depends a lot more on the set of initially selected nodes.

When applying the threshold model, highly similar results were achieved compared to the SI model. The resulting trend curves can be seen in Appendix C in Figure 2.36 to 2.39.

Conclusions

Applying diffusion models on snapshot based networks, created on the average code review length, can give an estimation of the lower bounds for an information to spread through a code review network. Looking at a hypothetical information started to spread in January 2018, only a few month are needed until more than half of all network nodes are reached. However, it should be impossible to derive a point in time with full coverage of informed nodes, since the networks are constantly evolving and new nodes are appearing steadily. Information tend to spread faster through code review in the Linux community compared to the BSD community. Furthermore, which nodes start to spread information plays a more critical role in BSD than in Linux.

Contact Sequence

At last, we created temporal networks as contact sequence based on the exact timestamps of the code review data as described in Section 2.4.3. Over the year 2018 45801 interactions took place at 15314 unique points in time for the BSD project and 687185 interactions on 355084 time points for the Linux kernel.

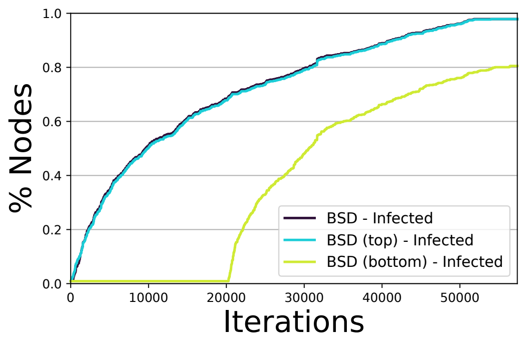


Figure 2.28: Diffusion Trend BSD SI model - contact sequence(2018)

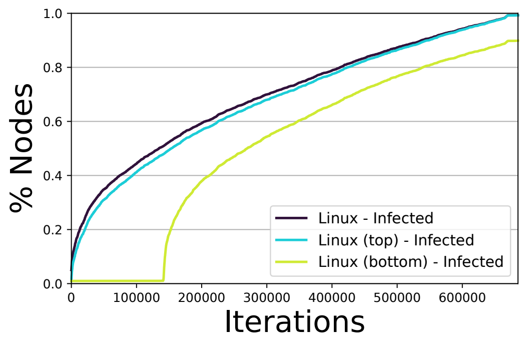


Figure 2.29: Diffusion Trend Linux SI model - contact sequence(2018)

Currently NDlib only supports the application of SI based models on contact sequences. Also, the automated multiple execution of a temporal model does not work for now. Since furthermore the execution of SI models on contact sequences needs considerable more time compared to the previously executed models, we only executed some spot tests with random selected initial infected nodes and achieved relatively similar results on multiple executions.

Figure 2.28 and Figure 2.29 both show an example of an execution with randomly selected 1% nodes in comparison to initial infected nodes selected, based on their interconnectivity as before. At the end of the year 97% nodes of the BSD network become infected for initial nodes selected at random or based on the highest node degree. In case of Linux 99% of all nodes get infected at the end of the year. When selecting the bottom 1% nodes based on their node degree, only 80%, respectively 90% are reached. 50% of all network nodes are infected after around 3 and a half month in both cases (01/01/18 - 03/14/18 resp. 01/01/18 - 03/22/18) when selecting the nodes with highest degree. When selecting the nodes with the lowest degree, 2 to 2 and a half month (06/07/18 - 08/07/18 resp. 03/13/18 - 05/29/18) This behavior is similar to the SI model previously run on snapshot networks.

Again, we executed the model varying the infection probability. The results can be seen in Figure 2.30 and Figure 2.31. As initial infection set we selected the 1% nodes with the highest node degree on the corresponding static network. When varying β the percentage of total infected nodes drops from 98% for $\beta = 1$ over 98% for $\beta = 0.75$ and 96% for $\beta = 0.5$ to 88% for $\beta = 0.25$. In contrast for Linux the average infection rate only drops from 99% for $\beta = 1$ over 99% for $\beta = 0.75$ and 96% for $\beta = 0.5$ to 90% for $\beta = 0.25$. Varying β has less of an effect compared to the model execution on snapshot graphs.

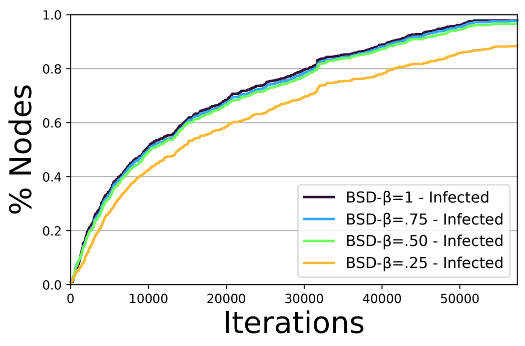


Figure 2.30: Varying beta BSD SI model - contact sequence(2018)

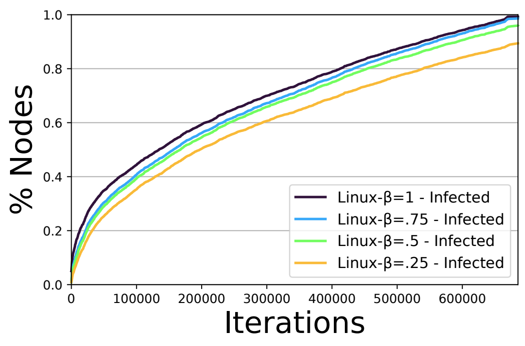


Figure 2.31: Varying beta Linux SI model - contact sequence(2018)

Varying the percentage of initial infected nodes based on their node degree did not affect the infection curves. Therefore, the curves with initial infected nodes based on the highest degree can be seen as the upper bound of infection spread. The corresponding plots can be found in Appendix D as Figure 2.40 and Figure 2.41.

Conclusions

Contact sequences give the most realistic view of communication behavior. This leads to higher run times, especially noticeable when working with larger graphs. The execution furthermore shows that using snapshots networks based on the average code review length tend to provide a good estimation of contact sequences.

One problem regarding the contact sequences created in our way, is the fact that the iterations are not necessarily uniformly distributed over time, since periods of time can have a different message exchange rate than others.

Interval Graph

Using an interval graph to construct a code review network seems to be the best suited approach. When the patch is posted a communication channel opens and information can be exchanged between all participants. We assume the duration of patch discussion plays a non negligible role in terms of information spreading and knowledge diffusion. Unfortunately using NDlib the construction of a network did not converge in reasonable time or produced unresolvable errors.

2.7 Discussions and Limitations

Since this is the first case study on information diffusion in code review and our primary goal was to show the viability of using contagion models on code review data, we did no classical validation of the measurement results except for manual spot testing during development. Nevertheless, we want to discuss the results and show limitations of our proposed measurement approach and the executed case study. In the following, possible problems and their solutions regarding the measurement method, our selected models, the data underlying the case study and their preparation will be discussed.

2.7.1 Limitations of Case Study Results

Both selected cases study subjects represent large open source operating system projects. This of course affects the generalizability of our findings regarding the information spread in those analyzed projects. We expect that the properties of information spread differ significantly in proprietary development in industrial context. But also matured open source projects not related to operating systems using other processes for code review or differing in project size will probably behave unalike. However, this should not affect the general implications of applying dynamic models of information diffusion on code review networks. If a code review network of a project can be created, it is possible to examine it by apply the proposed measurement model.

While we initially collected data of a broader time span, we ended up only analyzing the year of 2018. Since the temporal resolution of one year seems plenty long for one particular information to spread, the analysis of all collected data could possibly give insights in changes of development practices.

Furthermore, we did not vary the temporal resolution besides basing it on the average code review length and only started the model simulations at the beginning of 2018. Better sampling approaches with varied time frames and starting points are needed to achieve more convincing results regarding information flow.

Regarding the used data sources we assume that Patchwork, Phabricator and Bugzilla contain all relevant patches of the projects. Nevertheless, it can not be ruled out that patches are contributed to the projects over unforeseen ways. It would for instance be possible that a contributor mails a patch directly to a maintainer in a private email, who then reviews and applies the patch. However, in general both communities reinforce the public discussion of submitted patches. Regarding Bugzilla, it is possible that bug reports containing patches did not set the keyword [PATCH] according to the projects guidelines. No attempt was made in this thesis to find those patches. Moreover, due to the highly distributed development of the Linux kernel, it was not possible to cover all subsystems, especially missing projects hosted on freedesktop.org. We assume the created networks are representative for the Linux development community, although it can not be ruled out that excluded subsystems differ in regard to their information spread. In context of linux kernel development research, it could be worth investigating further differences between subsystems regarding the possible information flow.

While a basic account matching was done during data preparation, this step can surely be refined. The best results can only be reached when examining the results manually afterwards. Since this process is tedious and still complete correctness can not be guaranteed, we only performed manual spot tests to secure acceptable results. For projects primarily using centralized modern code review tools, this step is a lot easier or can even be skipped completely.

We did not exclude bots before constructing our social network. While it can be argued that those are not part of real human social communication, we state they play a non negligible role in terms of information spreading. For example when informing developers about automatically detected security problems.

Our networks are constructed by simple means, no distinctions were made whether someone submits or reviews code. Also for now, only comments on a patch were considered as communication data. However, modern Code review tools like Phabricator allow to use additional information like the acceptance or rejection of a patch. Furthermore, the expansion to include software development artifacts as analysis base can provide a more comprehensive picture.

When executing the threshold model, results highly similar to the SI model were achieved. We did not elaborate the selection of suited individual thresholds any further. It is likely that our selected thresholds being rather low in general have led to such similar results. Besides refining model parameters, the selection of models itself can probably lead to more meaningful results.

2.7.2 Limitations of Selected Diffusion Models

A model is always an abstraction of reality, but some are more capable of capturing it than others. Information is not diseases and while simple models like SI can give rough estimations, recent research is trying to find more suitable approaches. For example Min and San Miguel (2018) record that the adoption of information is a highly individual process and therefore propose a combined model where simple contagion can trigger a process of complex contagion. Milli et al. (2018a) differentiate active and passive diffusion to respect the influence of individual choices in the active adoption of content versus a passive infection like in disease outbreaks. They propose a mixed model to describe a process where an individual evaluates its own likelihood to adopt content after it has been exposed to a sufficient peer pressure. However, this referenced work serves only as an example, since the research on information diffusion models is still a rapidly evolving field.

2.7.3 Limitations of Measurement Model

As already mentioned, the selection of suitable modeling approaches is a key problem when measuring information diffusion. Nevertheless, unrelated to the selected model, our approach can only measure the information flow inherent to the underlying data. While undoubtedly a massive amount of information, respectively knowledge, can spread during a code review, it can also be transmitted over side channels. Classical models are based on the closed world assumption. Actors only gain information from other nodes of the same network, which is not the case in reality. A study of Myers, Zhu, and Leskovec (2012) has shown when looking at Twitter as an example, only 71% of the information volume can be attributed to network diffusion, while the remaining volume occurs due to factors and events outside the network.

One solution could be to apply the same approach on general social developer networks based on broader communication data. While this could be better at capturing the general information diffusion of a project it should be taken into account that unwanted non-technical discussion has to be filtered out. The analysis of code review networks therefore ensures that mostly communication regarding technical matters is considered. Nonetheless, using broader communication data still can not capture information exchanged at personal meetings during lunch, at work or at a conference.

Recent work in the field of opinion dynamics started to take this factor into account. Those models furthermore allow to investigate how different types of information can spread, since for now we only took the general spreading capabilities of a network into account, concerning a generic piece of information, while in reality most often multiple types of information are multiplexed in one message.

At last we are for now only measuring the flow of information. Information not necessarily transforms into knowledge once it is received. There are various reasons for this: the information is not relevant to everyone, it may be too complex, or one could simply forget it after some time. While our approach provides a possible upper bound for the transmission of knowledge, further efforts have to be made to measure the real transmission of knowledge during software development.

2.8 Conclusion

In this thesis, we gave an introduction to information diffusion in code review and took an initial step in measuring it. Covering the research question "How can the spread of information in code review networks be measured using models of information diffusion?" we showed a general measurement model to analyze code review data with models of information diffusion.

Thereby we presented the essential process steps needed for measurement: collecting and preparing code review data, creating a social network and applying a contagion model on it. Regarding the network creation we showed how different code review networks of varying temporal accuracy can be constructed. Using a minimum example, we furthermore showed how time information missing in a network can distort the outcome of diffusion processes in general, to illustrate the need of temporal modeling when analyzing information diffusion in code review. Moreover we presented two simple model classes of information diffusion, namely the SI and Threshold model. To the best of our knowledge this is the first work describing how code review processes can be modeled using temporal networks and analyzed using means of information diffusion.

We showed the viability and usefulness of our measurement model using case study research. We were able to compare the information diffusion processes of FreeBSD and the Linux kernel as two large and matured open source projects in regard to the possible information throughput by constructing and analyzing different types of code review networks. Depending on the model we showed which fraction of nodes can be reached at total, how many hops, respectively time is needed to inform a specified fraction of nodes, how the curve characteristics can vary and how it matters which individuals initially start to spread information. While our case study subjects behave mostly similar regarding their possible information flow regarding time and reached nodes, it plays a much more crucial role who initially spreads information in case of FreeBSD than in Linux. Moreover, huge differences result when comparing the outcomes in regard to the underlying network model. The contrast of resulting spreading processes empathize the need of temporal modeling when analyzing information spread in code review.

Overall we believe the application of diffusion models on code review networks are a useful method to analyze information and knowledge spreading, especially when using temporal networks as a foundation. Nevertheless the field of information diffusion in code review, respectively in software engineering in general, still provides plenty of research opportunities.

2.8.1 Future Work

Unfortunately, up to now no standard tooling has been established to analyze diffusion in social networks. While NDlib provided a good starting point for research, it still can be optimized, especially regarding the analysis of large networks. Simulating models on temporal networks is not multi threaded for now, we were not able to create interval graphs except for small data sets and found minor errors in the visualization framework. Furthermore the documentation of DynetX as NDlibs basis can be extended.

Since the outcomes of the applied models for our two selected open source operating systems are highly similar, it would be interesting to apply the same models on industry grade projects of similar size, or on smaller open source projects. Also changing the subject to smaller sub communities could bring valuable results, since plenty of information is only relevant to a specific audience. Starting with smaller research subjects one could further investigate to which degree information transforms into knowledge. In addition, finding more suited models and sensible parameter settings to better depict reality is an important step needed to analyze the spread of information and knowledge in code review.

Separate of code review, our proposed measurement model can furthermore be applied to more general communication data arising during software development. Finally we hope that research in modeling information diffusion processes happening in various fields can further be unified.

Appendix A Fruchterman-Reingold Layout

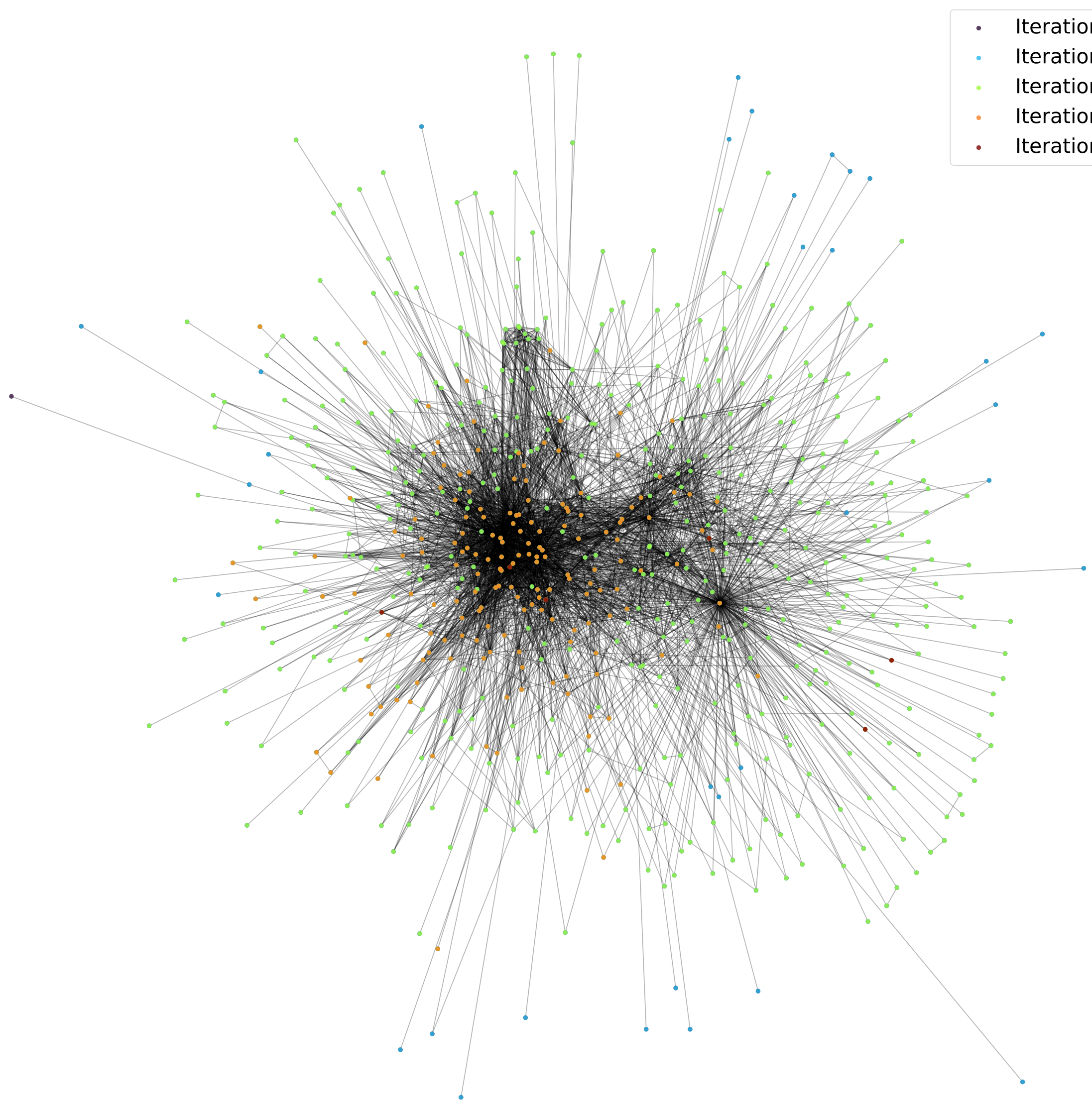


Figure 2.32: Diffusion Trend BSD - Fruchterman–Reingold layout

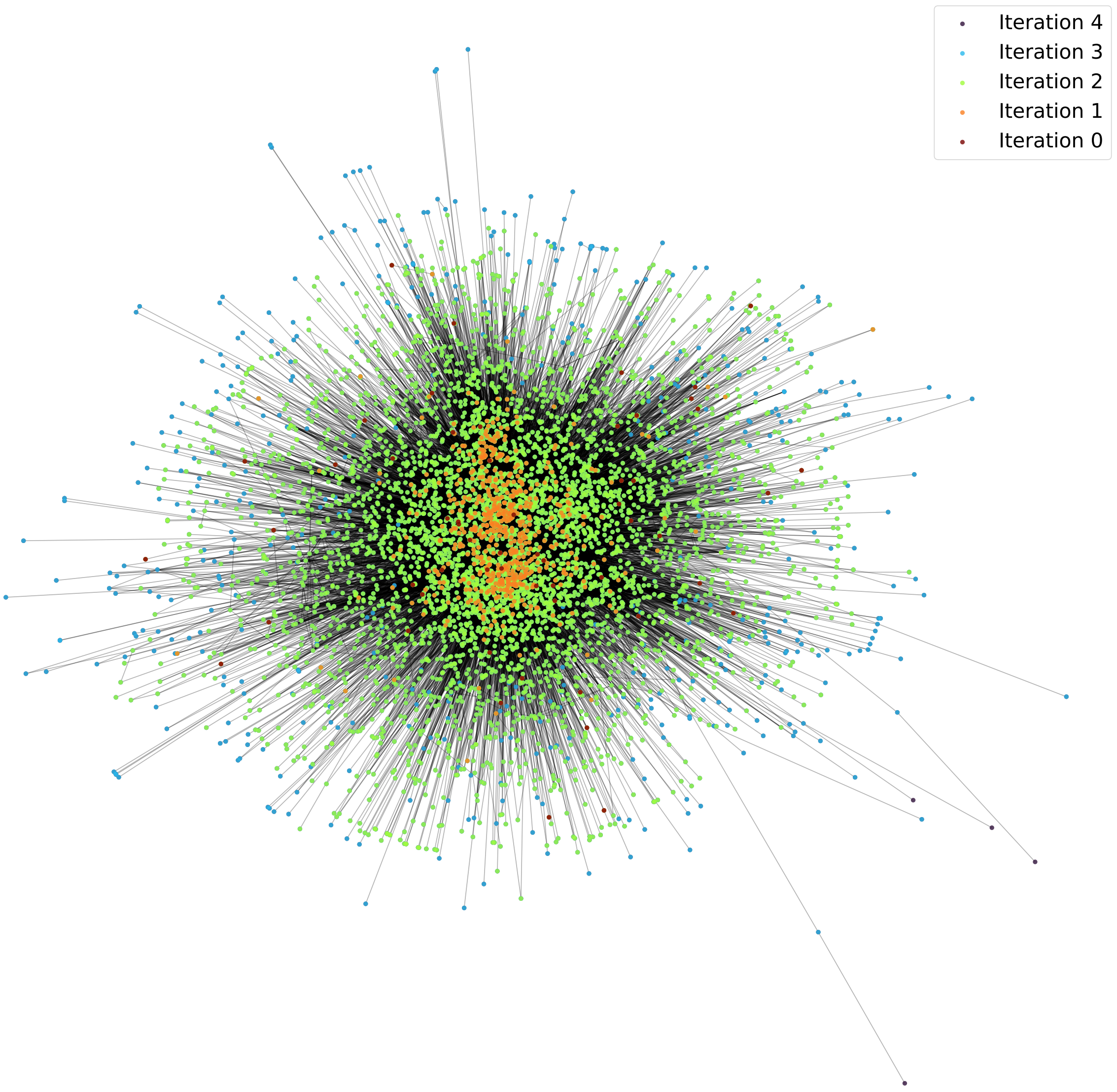


Figure 2.33: Diffusion Trend Linux - Fruchterman–Reingold layout

Appendix B Threshold Model - Comparison per Time Interval

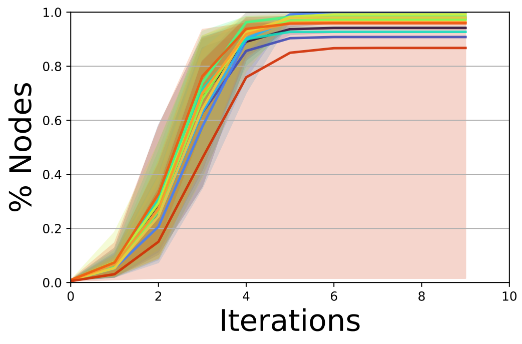


Figure 2.34: Diffusion Trend BSD (per month - 2018)

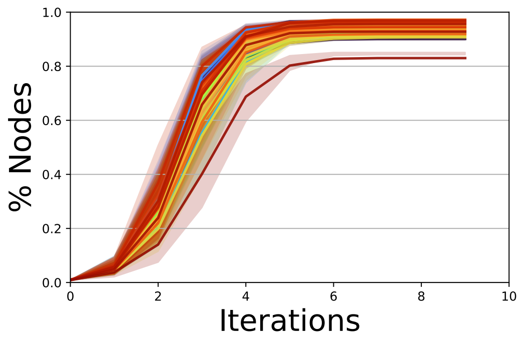


Figure 2.35: Diffusion Trend Linux (per week - 2018)

Appendix C   Threshold Model - Snapshot Based

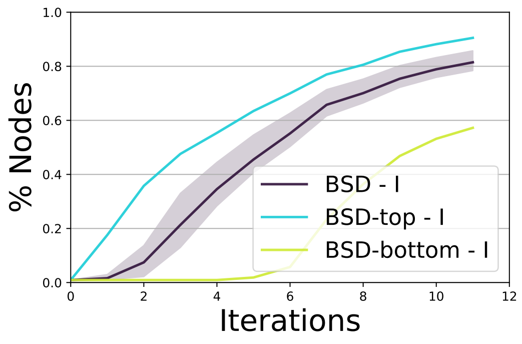


Figure 2.36: Diffusion Trend BSD
Threshold snapshot based (2018)

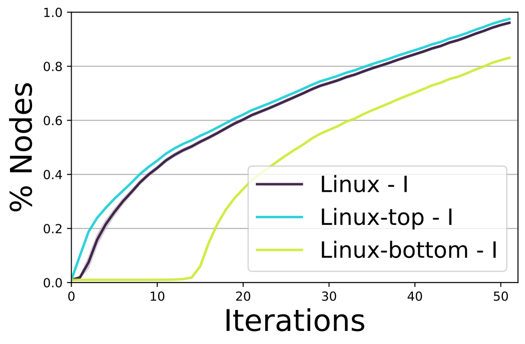


Figure 2.37: Diffusion Trend Linux
Threshold snapshot based(2018)

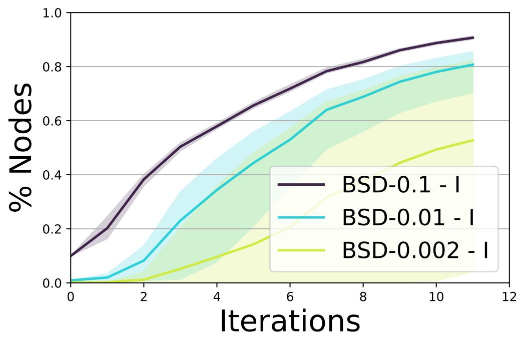


Figure 2.38: Varying the initial infection percentage BSD
Threshold snapshot based (2018)

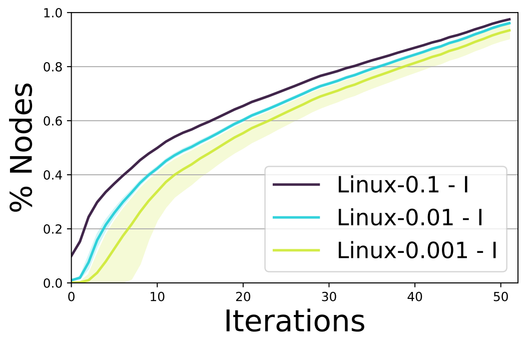


Figure 2.39: Varying the initial infection percentage Linux
Threshold snapshot based(2018)

Appendix D SI Model - Contact Sequence

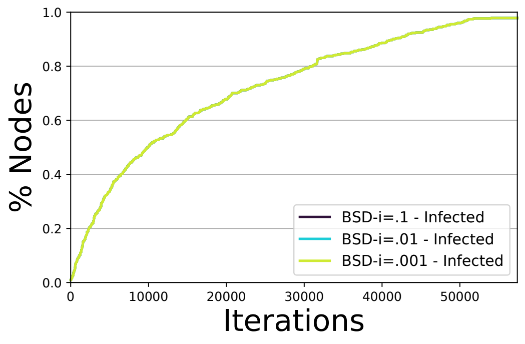


Figure 2.40: Varying the initial infection percentage BSD
SI Contact Sequence(2018)

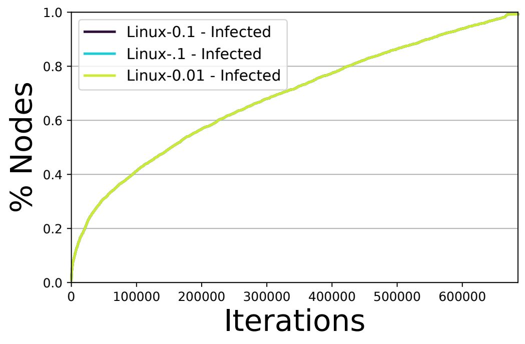


Figure 2.41: Varying the initial infection percentage Linux
SI Contact Sequence(2018)

References

Amirfallah, A., Trautsch, F., Grabowski, J., & Herbold, S. (2019). A systematic mapping study of developer social network research. *ArXiv.org (Cornell University Library), arXiv:1902.07499*. Retrieved from https://arxiv.org/abs/1902.07499

Anderson, R. M., & May, R. M. (1979). Population biology of infectious diseases: Part I. *Nature*, *280*(5721), 361–367. doi:10.1038/280361a0

Bacchelli, A., & Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering* (pp. 712–721). doi:10.1109/ICSE.2013.6606617

Baum, T., Liskin, O., Niklas, K., & Schneider, K. (2016a). A faceted classification scheme for change-based industrial code review processes. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)* (pp. 74–85). IEEE. doi:10.1109/QRS.2016.19

Baum, T., Liskin, O., Niklas, K., & Schneider, K. (2016b). Factors influencing code review processes in industry. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 85–96). doi:10.1145/2950290.2950323

Bird, C., Gourley, A., Devanbu, P., Gertz, M., & Swaminathan, A. (2006). Mining email social networks. In *Proceedings of the 2006 International Workshop on Mining Software Repositories* (pp. 137–143). doi:10.1145/1137983.1138016

Bosu, A., & Carver, J. C. (2014). Impact of developer reputation on code review outcomes in oss projects: An empirical investigation. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (pp. 1–10). doi:10.1145/2652524.2652544

Centola, D., & Macy, M. (2007). Complex contagions and the weakness of long ties. *American journal of Sociology*, *113*(3), 702–734. doi:http://doi.org/10.1086/521848

Cha, M., Haddadi, H., Benevenuto, F., & Gummadi, K. P. (2010). Measuring user influence in twitter: The million follower fallacy. In *fourth international AAAI conference on weblogs and social media* (pp. 10–17). doi:10.1145/2897659.2897663

Corbet, J. (2019). 5.3 kernel development cycle statistics. https://lwn.net/Articles/798505/. Retrieved: March 2020.

Fagan, M. E. (1976). Design and code inspections to reduce errors in program development. *IBM Syst. J.*, *15*(3), 182–211. doi:10.1147/sj.153.0182

FreeBSD Documentation Project, T. (2019). Committer's guide. https://www.freebsd.org/doc/en_US.ISO8859-1/articles/committers-guide/. Retrieved: March 2020.

FreeBSD Documentation Project, T. (2020). FreeBSD handbook. https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/nutshell.html. Retrieved: March 2020.

freedesktop.org. (2019). Patchwork. https://gitlab.freedesktop.org/patchwork-fdo/patchwork-fdo. Retrieved: March 2020.

Fruchterman, T. M., & Reingold, E. M. (1991). Graph drawing by force-directed placement. *Software: Practice and experience*, *21*(11), 1129–1164. doi:10.1002/spe.4380211102

Goffman, W., & Newill, V. A. (1964). Generalization of epidemic theory: An application to the transmission of ideas. *Nature*, *204*(4955), 225–228. doi:10.1038/204225a0

Granovetter, M. (1978). Threshold models of collective behavior. *American journal of sociology*, *83*(6), 1420–1443. doi:10.1086/226707

Granovetter, M. (1983). The strength of weak ties: A network theory revisited. *Sociological Theory*, *1*, 201–233. doi:10.2307/202051

Guille, A., Hacid, H., Favre, C., & Zighed, D. A. (2013). Information diffusion in online social networks: A survey. *ACM SIGMOD Record*, *42*(2), 17–28. doi:10.1145/2503792.2503797

Hamasaki, K., Kula, R. G., Yoshida, N., Cruz, A. C., Fujiwara, K., & Iida, H. (2013). Who does what during a code review? datasets of oss peer review repositories. In *2013 10th Working Conference on Mining Software Repositories* (pp. 49–52). IEEE. doi:10.1109/MSR.2013.6624003

Herrera, M., Armelini, G., & Salvaj, E. (2015). Understanding social contagion in adoption processes using dynamic social networks. *PLOS ONE*, *10*(10), 1–25. doi:10.1371/journal.pone.0140891

Holme, P., & Saramäki, J. (2012). Temporal networks. *Physics reports*, *519*(3), 97–125. doi:10.1016/j.physrep.2012.03.001

Hubbard, J., Lawrance, S., & Linimon, M. (2020). Contributing to freebsd. https://www.freebsd.org/doc/en_US.ISO8859-1/articles/contributing/article.html. Retrieved: March 2020.

Kempe, D., Kleinberg, J., & Tardos, É. (2003). Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 137–146). doi:10.1145/956750.956769

Kermack, W. O., & McKendrick, A. G. (1927). A contribution to the mathematical theory of epidemics. *Proceedings of the royal society of london. Series*

A, Containing papers of a mathematical and physical character, *115*(772), 700–721. doi:10.1098/rspa.1927.0118

kernel development community, T. (n.d.-a). How the development process works. https://www.kernel.org/doc/html/latest/process/2.Process.html. Retrieved: March 2020.

kernel development community, T. (n.d.-b). Submitting patches: The essential guide to getting your code into the kernel. https://www.kernel.org/doc/html/latest/process/submitting-patches.html. Retrieved: March 2020.

Liben-Nowell, D., & Kleinberg, J. (2008). Tracing information flow on a global scale using internet chain-letter data. *Proceedings of the National Academy of Sciences*, *105*(12), 1–15. doi:10.1073/pnas.0708471105

Milli, L., Rossetti, G., Pedreschi, D., & Giannotti, F. (2018a). Active and passive diffusion processes in complex networks. *Applied network science*, *3*(1), 42. doi:10.1007/s41109-018-0100-5

Milli, L., Rossetti, G., Pedreschi, D., & Giannotti, F. (2018b). Diffusive phenomena in dynamic networks: A data-driven study. In S. Cornelius, K. Coronges, B. Gonçalves, R. Sinatra, & A. Vespignani (Eds.), *Complex Networks IX* (pp. 151–159). doi:10.1007/978-3-319-73198-8_13

Min, B., & San Miguel, M. (2018). Competing contagion processes: Complex contagion triggered by simple contagion. *Scientific reports*, *8*(10422), 1–8. doi:10.1038/s41598-018-28615-3

Myers, S. A., Zhu, C., & Leskovec, J. (2012). Information diffusion and external influence in networks. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 33–41). doi:10.1145/2339530.2339540

NetworkX developers. (2020). NetworkX - network analysis in python. https://networkx.github.io/. Retrieved: March 2020.

Newman, M. (2018). *Networks* (2nd ed.). Oxford university press.

NumPy developers. (2020). Numpy. https://numpy.org/. Retrieved: March 2020.

pandas development team, T. (2020). Pandas. https://pandas.pydata.org. Retrieved: March 2020.

Patchwork Developers. (2018). The rest api. https://patchwork.readthedocs.io/en/latest/api/rest/. Retrieved: March 2020.

Patchwork Developers. (2019). Patchwork. https://github.com/getpatchwork/patchwork. Retrieved: March 2020.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., … Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830. doi:10.5555/1953048.2078195

Phacility, Inc. (n.d.). Conduit - modern methods. https://reviews.freebsd.org/conduit/. Retrieved: March 2020.

Project Jupyter. (2020). Jupyter. https://jupyter.org/. Retrieved: March 2020.

Rigby, P. C., & Bird, C. (2013). Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (pp. 202–212). doi:10.1145/2491411.2491444

Rigby, P. C., German, D. M., Cowen, L., & Storey, M.-A. (2014). Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *23*(4), 1–33. doi:10.1145/2594458

Rogers, E. M. (2003). *Diffusion of innovations* (5th ed.). Simon and Schuster.

Rossetti, G. (2017). DyNetx - dynamic network library. https://dynetx.readthedocs.io/en/latest/index.html. Retrieved: March 2020.

Rossetti, G., & Cazabet, R. (2018). Community discovery in dynamic networks: A survey. *ACM Computing Surveys*, *51*(2), 1–37. doi:10.1145/3172867

Rossetti, G., Milli, L., Rinzivillo, S., Sîrbu, A., Pedreschi, D., & Giannotti, F. (2018). NDLIB: A python library to model and analyze diffusion processes over complex networks. *International Journal of Data Science and Analytics*, *5*(1), 61–79. doi:10.1007/s41060-017-0086-6

SeatGeek. (2020). Fuzzywuzzy. https://github.com/seatgeek/fuzzywuzzy. Retrieved: March 2020.

Shakarian, P., Bhatnagar, A., Aleali, A., Shaabani, E., & Guo, R. (2015). *Diffusion in social networks.* doi:10.1007/978-3-319-23105-1

Xu, Y., & Zhou, M. (2018). A multi-level dataset of linux kernel patchwork. In *Proceedings of the 15th International Conference on Mining Software Repositories* (pp. 54–57). doi:10.1145/3196398.3196475

Yang, X. (2014). Social network analysis in open source software peer review. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 820–822). doi:10.1145/2635868.2661682

Yang, X., Kula, R. G., Erika, C. C. A., Yoshida, N., Hamasaki, K., Fujiwara, K., & Iida, H. (2012). Understanding oss peer review roles in peer review social network (PeRSoN). In *2012 19th Asia-Pacific Software Engineering Conference* (Vol. 1, pp. 709–712). IEEE. doi:10.1109/APSEC.2012.63

Yang, X., Yoshida, N., Kula, R. G., & Iida, H. (2016). Peer review social network (PeRSoN) in open source projects. *IEICE Transactions on Information and Systems*, *99*(3), 661–670. doi:10.1587/transinf.2015EDP7261

Yin, R. K. (2017). *Case study research and applications: Design and methods* (6th ed.). Sage publications.

Zhan, X.-X., Hanjalic, A., & Wang, H. (2019). Information diffusion backbones in temporal networks. *Scientific reports*, *9*(1), 1–12. doi:10.1038/s41598-019-43029-5

Zhang, H., Mishra, S., Thai, M. T., Wu, J., & Wang, Y. (2014). Recent advances in information diffusion and influence maximization in complex social networks. *Opportunistic Mobile Social Networks*, *37*(1.1), 37. doi:10.1201/b17231