Impact of Developer Reputation on Code Review Outcomes in OSS Projects: An Empirical Investigation

Amiangshu Bosu, Jeffrey C. Carver
Department of Computer Science
University of Alabama
Tuscaloosa, AL, USA
asbosu@ua.edu, carver@cs.ua.edu

ABSTRACT

Context: Gaining an identity and building a good reputation are important motivations for Open Source Software (OSS) developers. It is unclear whether these motivations have any actual impact on OSS project success. Goal: To identify how an OSS developer’s reputation affects the outcome of his/her code review requests. Method: We conducted a social network analysis (SNA) of the code review data from eight popular OSS projects. Working on the assumption that core developers have better reputation than peripheral developers, we developed an approach, Core Identification using K-means (CIK) to divide the OSS developers into core and periphery groups based on six SNA centrality measures. We then compared the outcome of the code review process for members of the two groups. Results: The results suggest that the core developers receive quicker first feedback on their review request, complete the review process in shorter time, and are more likely to have their code changes accepted into the project codebase. Peripheral developers may have to wait 2 - 19 times (or 12 - 96 hours) longer than core developers for the review process of their code to complete. Conclusion: We recommend that projects allocate resources or create tool support to triage the code review requests to motivate prospective developers through quick feedback.

Categories and Subject Descriptors
K.6.3 [Management of Computing and Information Systems]: Software Management—Software development, Software process

General Terms
Measurement, Security, Human Factors

Keywords
code review, open source, social network analysis, peer impression, network structure

1. INTRODUCTION

The success of an Open Source Software (OSS) project is affected by the contributions of each individual developer. One of the primary motivations for many OSS developers is the desire to gain an identity and build a good reputation [22]. This motivation drives the amount of effort an OSS developer devotes to the OSS project [20] and helps ensure that an OSS developer invest his/her efforts to keep the project on track to be successful [26]. While a good reputation is an important goal for individual OSS developers, it is not clear what impact a developer’s reputation has on project success. If having a good reputation increases the value of an OSS developer’s project contributions, then OSS projects that facilitate reputation building should be more successful.

There are different ways to contribute to an OSS project — e.g., posting defects, participating in mailing-list discussions, submitting bug fixes, providing user support, reviewing code, and making code changes. Contributions related directly to code (i.e. making code changes and reviewing code changes) are among the most important types of contributions, because delivering quality software requires contributors to make quality code changes. To ensure the inclusion of only high-quality code into the project codebase, many mature OSS projects have adopted code review as a mandatory quality assurance gateway, thereby elevating the importance of code review. If a developer’s reputation can be shown to have a positive impact on the outcome of code review, then we have evidence of the value of developer’s reputation on the project outcome as a whole. Therefore, the objective of this study is to identify how an OSS developer’s reputation affects the outcome of his/her code review requests. Specifically, we explore whether an OSS developer’s reputation affects the following aspects of code review: 1) feedback time, 2) review interval, 3) acceptance of code, and 4) number of patches before a code is accepted.

The structure of OSS development communities has often been described as a core-periphery structure [12, 28, 40], with a small number of core developers and a larger set of peripheral developers. The small set of core developers are those who have been involved with the OSS project for a relatively long time and make significant contributions to guide the development and evolution of the project [40]. The larger set of peripheral developers occasionally contribute to the project [40], mostly interact with the core developers, and rarely interact with other peripheral developers [21]. Due to their significant contributions and higher level of interactions, core developers are the most reputed contributors in an OSS community [40]. Therefore, the core-periphery distinction should be a good proxy for reputation.

In this study, we use social network analysis (SNA) of code review interactions to divide the OSS developers into core and periphery groups. In OSS projects that require all code to undergo peer review, the number of changes a developer has submitted for review and the number of changes s/he has reviewed serve as proxies for his/her level of involvement in the project. Therefore, in
these types of projects, analysis of code review interactions can reveal the most influential developers (i.e. core) of an OSS project. Moreover, the results of our prior studies also support using code review social networks to identify the most reputed users. First, the results of a large scale survey of OSS developers showed that code review interactions are highly effective in building positive reputation among the review participants [9]. Subsequently, we found that due to direct interactions between the developers and lower network centralization, code review social networks have the most favorable characteristics to support building mutual reputation [10].

We create code review social networks of eight popular OSS projects using data mined from Gerrit code review repositories. We develop a novel Core Identification using K-means (CIK) clustering approach based on six SNA centrality measures to divide the OSS developers into core and periphery groups. We then validated the correctness of the new CIK approach. Working on the assumption that core developers have a better reputation than peripheral developers, we compared the code review outcomes of core developers to the code review outcomes of the peripheral developers to investigate the effects of reputation. The main contributions of this work are:

- empirical evidence regarding the effect of OSS developers’ reputation on code review outcome,
- an overview of OSS social networks based on code review interactions, and
- a novel approach to group OSS developers into core and periphery.

The rest of the paper is organized as follows. Section 2 provides background about code review, and social network analysis. Section 3 introduces the research questions. Section 4 describes our novel approach to identify the core nodes. Section 5 describes the analysis and results. Section 6 discusses the implications of the results. Section 7 addresses the threats to validity. Finally, Section 8 concludes the paper.

2. BACKGROUND

This section provides background information on three topics relevant to this study: code review, social network analysis, and core-periphery structure in OSS projects.

2.1 Contemporary Code Review

Code review (a.k.a. peer code review), the process of analyzing code written by a teammate (i.e. a peer) to judge whether it is of sufficient quality to be integrated into the main project codebase, has recently been adopted by many mature successful OSS projects. According to Bacchelli and Bird [1], contemporary code reviews are more informal and more common than the Fagan-style inspection [15]. In OSS and commercial organizations, code reviews are also increasingly supported by tools [34].

One of the prominent code review tools is Gerrit\(^1\). Gerrit captures the following detailed information about the code review that is required for our study. Developers submit a patchset (i.e. all files added or modified in a single revision) to a Gerrit repository for review. The Gerrit interface displays the changes side-by-side, allows the reviewers to insert inline comments, and allows the author to respond to those comments. Gerrit documents all of these interactions. If reviewers request changes, the author can make those changes and upload a new patchset. This process repeats until the reviewers approve the patchset, which can then be merged into the main project branch.

Projects included in this study require that each code change pass through a mandatory code review before inclusion in the project codebase. Therefore, the number of ‘Merged’ review requests is a measure of the number of commits by an OSS developer that were integrated into the project codebase. The sum of the number of ‘Merged’ and ‘Abandoned’ requests is a measure of the total number of patchsets submitted by a OSS developer. These two measures allow us to use code review data to calculate the code contributions of a developer.

2.2 Social Network Analysis

A social network is a theoretical construct that represents the relationships between individuals. These networks are typically modeled as a graph in which nodes represent individuals and edges represent the relationship between individuals. SNA identifies social relationships and interaction patterns among individuals in a community based on the idea that these patterns represent an important aspect of the lives of those individuals [17]. These social interaction patterns also affect some important features of a community (e.g., efficiency when performing a task, group dynamics, and leadership) [39]. Therefore, SNA is a common approach for identifying and studying relationships in a variety of domains, e.g., sociology, biology, anthropology, communication studies, information science, organizational studies, and psychology.

Software engineering researchers have used SNA techniques to understand different types of developer interactions in OSS projects. First, using code commit interactions, studies show that developers who work on the same file are more likely to interact in the mailing-list [5], and SNA techniques can identify the influence of a sponsoring company on the development process, release management, and leadership turnover [27]. Second, using bug fixing interactions, studies found decentralized communication patterns in larger projects [11] and a stable core-periphery structure with decreasing group centralization over time [24]. Third, using mailing-list interactions, studies found developers having higher status than non-developers [4] and the core developers having disproportionately large share of communication with the peripheral developers [31]. The application of SNA techniques in our study differs from those prior studies in two key ways: 1) using source data from a different types of interaction (i.e. code review), and 2) using SNA techniques for measuring developer reputation.

2.3 Core-Periphery Structure in OSS Communities

A social network that exhibits the Core-Periphery structure has a set of densely connected core nodes and a set of more loosely connected periphery nodes [7]. For example, Figure 1 shows a toy network of 10 nodes with a core-periphery structure. The four central nodes (i.e. 1, 2, 3, and 4) have connections with each other and forms the dense core. The remaining six nodes are loosely connected with the core nodes and do not have edges between them.

Several studies has focused on characterizing the core-periphery structures of OSS development communities. Fielding was the first to describe the core developer group and their roles in the Apache project [16]. Later studies found that while the core contains only 3-25% of the developers, these developers contribute 40-90% of the code (more details in Section 4.4) [12,21,23,28,36]. Although, the number of core developers in a project may not vary much, the members belonging the core change over time [36]. However, frequent core membership turnover may not be good for the sustainability of an OSS project. Long-term core members may have detailed knowledge about the design of the project and are often able to perform maintenance tasks to reduce structural complex-

\(^1\) https://code.google.com/p/gerrit/
ity of the project [38]. While most of the prior studies focused on estimating the number of core developers, their contributions, and their roles in the project, we focus on developing a method to identify the core developers and how their code review outcomes differ from those of the peripheral developers.

3. RESEARCH HYPOTHESES

The high-level hypothesis of this study is, “A Developer’s reputation affects the outcome of his/her code review requests”. Specifically, we use the core-periphery distinction as our measure of reputation. That is, developers in the core have a better reputation than those in the periphery. Therefore, this hypothesis really suggests that aspects of the code review process are affected by whether the code author is a member of the core or a member of the periphery. The following subsections decompose this high-level hypothesis into four detailed hypotheses.

3.1 First Feedback Interval

We define First Feedback Interval as the amount of time from the submission of a code review request in Gerrit until the first real review comment (i.e. not an automated comment). There are several factors that suggest that core developers should have a shorter first feedback interval than the peripheral developers:

- They should know which teammates are most appropriate for reviewing a particular change and can add them as potential reviewers;
- Their position in the core group may lead reviewers to provide a less thorough, less time-consuming review; and
- Their prior interactions and good relationships may encourage other members to prioritize their code for review.

Therefore, we pose the following hypothesis:

**H1** Core developers have shorter first feedback interval than the peripheral developers.

3.2 Review Interval

Review Interval is the time from the beginning to the end of the review process [35]. We define the review process to be complete when the patchset is ‘Merged’ to the main project branch or is ‘Abandoned’. For similar reasons as listed in Section 3.1, core developer may have a shorter overall review interval. Moreover, due to their familiarity with the codebase and their prior interactions with the reviewers, core developers may be able to understand and make the suggested changes more quickly than the peripheral developers. Therefore, we pose the following hypothesis:

**H2** Core developers have shorter review interval than the peripheral developers.

3.3 Code Acceptance Rate

A developer’s Code Acceptance Rate is the ratio between the number of review requests submitted and the number ‘Merged’. Due to more familiarity about the project design, we suggest that core developers are able to write acceptable code more frequently than the peripheral developers. Moreover, their positions in the core help them influence the project development direction [19], to integrate their suggested features into the project more often. Therefore, we pose the following hypothesis:

**H3** Core developers have higher code acceptance rate than the peripheral developers.

3.4 Number of Patchsets per Review Request

If a reviewer identifies a problem during code review, the author must upload a new patchset to fix that problem. The reviewer reviews the new patchset and either accepts it or requests further modifications. This process repeats until the reviewer is satisfied with the changes and agrees the code is ready to be merged. Because core developers have already made many code changes, they should be familiar with the project requirements, design, and coding guidelines. Therefore, they should be able to write code that can be merged in fewer tries (i.e. fewer patchsets). In addition, because they are in the core, other developers may approve their patchsets more quickly. Therefore, we pose the following hypothesis:

**H4** Core developers are able to get code changes accepted through lower number of patchsets than the peripheral developers.

4. CORE IDENTIFICATION USING K-MEANS (CIK)

The following subsections describe the collection and preparation of data, construction of the social networks, detection of core and peripheral developers, and evaluation of the approach.

4.1 Data Collection and Preparation

Following a similar approach as Mukadam et al. [29], we developed the *Gerrit-Miner* tool that can mine the publicly accessible code review data in a Gerrit repository. In January of 2014, we used *Gerrit-Miner* to mine the code review repositories of eight popular OSS projects (Table 2) with regular code review activity. We mined all completed code review requests (i.e. those marked as either ‘Merged’ or ‘Abandoned’). A manual inspection of the comments posted by some accounts (e.g., Qt Sanity Bot or BuildBot) suggested that those accounts were automated bots rather than humans. These accounts typically contain one of the following key-words: ‘bot’, ‘auto’, ‘CI’, ‘Jenkins’, ‘integration’, ‘build’, or ‘verifier’. Because we wanted only code review comments from actual reviewers, we excluded these bot accounts after a manual inspection confirmed that the comments were automatically generated.

4.2 Building Social Networks

Based on the mined data, we calculated the number of code review interactions between each pair of developers in each project. Using this data, we generated social network graphs as undirected, weighted graphs where nodes represent developers and edge weights represent the quantity of interactions between those developers. For
example, if Developer A has commented on 10 code review requests posted by Developer B and Developer B has commented on 15 code review requests posted by Developer A, the weight of the edge between their nodes would be 25. Conversely, if Developers A and B had not commented on each others’ code review requests, then no edge would exist.

Based on this data, Figure 2 shows the social network structure for two of the eight projects (LibreOffice and Chromium OS) as generated by Gephi [3]. Node size visualizes node degree (i.e. the number of attached edges). Node color indicates whether the node is in the core or periphery set (see Section 4.3). Edge widths are based on edge weights. To make the diagrams more readable, we removed low-weight edges and then excluded isolated nodes.

4.3 Core-Periphery Detection using the CIK Approach

While Borgatti and Everett’s statistical approach based on the density of links between nodes is the most common approach to distinguish core nodes from periphery nodes [7], it is not applicable in this study. Borgatti and Everett’s approach works well for social network with single core group (e.g. LibreOffice - left side of Figure 2). However, it does not work well for networks with multiple core groups (e.g. Chromium OS - right side of Figure 2). In fact, the authors suggest that networks with multiple cores be split into multiple networks and analyzed separately [14]. In practice, it may be difficult to divide the social networks of the OSS projects in our sample into separate networks with clearly distinguishable core and periphery. For example, the Chromium OS project has two clear groups (i.e. the big cluster on left and small cluster on right). However, there are several groups within the big cluster that are difficult to separate. In this graph, Borgatti and Everett’s approach is not useful.

Therefore, we developed a new approach for partitioning the OSS social networks into core and periphery based on centrality measures (i.e. measures of the relative importance of a node within a graph). Because core developers are typically the most important developers in an OSS project, we can use centrality measures to find the nodes that represent those core developers. There are a number of widely used centrality measures: degree, betweenness, closeness, eigenvector, PageRank, and eccentricity. Each measure calculates centrality in a different way and has a different interpretation of a central node. To include the best of all options and reduce the bias from any one centrality measure, we used the K-means clustering algorithm\(^2\) [25] to combine all six measures into one new measure. We call this approach Core Identification using K-means (CIK).

Table 1 provides an overview of the six centrality measures and their interpretation within an OSS social network. We used Gephi to calculate the six centrality measures for each node in each graph. Then, we used the SPSS implementation of the k-means clustering algorithm to partition the nodes into core and periphery groups based on the six centrality scores. Table 2 provides a description of the core-periphery partitions for the eight projects in this study.

4.4 Evaluation of the CIK Approach

We used a four-step approach to validate the appropriateness of the CIK approach. In Step 1, we compared the results of the CIK approach to the results of the Borgatti-Everett approach on a toy network and on a single-core network. In Step 2, we visually inspected the SNA networks to determine if core nodes identified by the CIK approach pass the ‘eye-test’ of belonging to a dense, cohesive core. In Steps 3 and 4, we compared the size and level of contributions of the core developers with the results from prior studies on OSS communities.

Comparison to Borgatti-Everett Algorithm

Using two networks, we compared the results of our approach to the results of the Borgatti-Everett approach on a toy network and on a single-core network. In Step 2, we visually inspected the SNA networks to determine if core nodes identified by the CIK approach pass the ‘eye-test’ of belonging to a dense, cohesive core. In Steps 3 and 4, we compared the size and level of contributions of the core developers with the results from prior studies on OSS communities.

\(^2\) K-means clustering partitions \(n\) observations into \(k\) mutually exclusive clusters in which each observation belongs to the cluster with the nearest mean. K-means treats each observation as an object having a location in space. It finds a partition in which objects within each cluster are as close to each other as possible and as far from objects in other clusters as possible.
Table 1: Social Network Centrality Measures

<table>
<thead>
<tr>
<th>Metric</th>
<th>Definition</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degree Centrality</td>
<td>Degree Centrality is the number of edges incident upon a vertex [18]. The degree centrality of a vertex v, for a given graph G, is defined as: ( C_D(v) = \deg(v) ).</td>
<td>Degree centrality is an indication of how many other developers each developer interacts with. Core developers usually interact with a large number of other developers resulting in a higher degree centrality.</td>
</tr>
<tr>
<td>Betweenness Centrality</td>
<td>The Betweenness Centrality of a vertex v is a measurement of the number of shortest paths traversing that vertex [18]. The betweenness centrality of vertex v in graph G is defined as: ( C_B(v) = \frac{\sum_{s \neq v \neq t} \delta_{st}(v)}{\sum_{s \neq t} \delta_{st}} ), where ( \delta_{st}(v) ) is the number of shortest paths from s to t going through v, and ( \delta_{st} ) is the total number of shortest paths between s and t.</td>
<td>Betweenness centrality can be interpreted as a measurement of the importance of a developer within a network, because developers with a high betweenness centrality are intermediate nodes for the communication of the other developers.</td>
</tr>
<tr>
<td>Closeness Centrality</td>
<td>The Closeness Centrality of a vertex v is a measurement of its proximity to the rest of the vertices in the network [37]. The higher the value, the closer that vertex is to the others (on average). The closeness centrality of vertex v in graph G is defined as: ( C_C(v) = \frac{1}{\sum_{t \neq v} d_C(v, t)} ), where ( d_C(v, t) ) is the minimum distance between v and t.</td>
<td>Closeness centrality gives an indication of how quickly a developer can reach the entire community.</td>
</tr>
<tr>
<td>Eigenvector Centrality</td>
<td>Eigenvector Centrality is a measure of the importance of a vertex in a network. It assigns relative scores to all vertices in the network based on the principle that connections to high-scoring vertices contribute more to the score of the vertex in question than equal connections to low-scoring vertices. Eigenvector centrality is defined as the principal eigenvector of the adjacency matrix defining the network. The defining equation of &quot;an eigenvector is λv = Av where A is the adjacency matrix of the graph, λ is a constant (the eigenvalue) and V is the eigenvector. The equation lends itself to the interpretation that a node that has a high eigenvector score is one that is adjacent to nodes that are themselves high scorers&quot; [6].</td>
<td>Eigenvector centrality gives an indication of a developer’s influence over the network.</td>
</tr>
<tr>
<td>PageRank Centrality</td>
<td>PageRank Centrality is based on Google’s PageRank algorithm. PageRank is a variant of Eigenvector Centrality and is based on the same basic concept. The key difference in PageRank is the use of a random surfer model based on a random decay factor (d - called damping factor, usually 0.85), which means that a random surfer would stop following links on the existing node with probability d and teleport to a new node. [32].</td>
<td>PageRank also gives an indication of a developer’s influence over the network.</td>
</tr>
<tr>
<td>Eccentricity</td>
<td>Eccentricity is a measure of how far a vertex is from the most distant vertex. A high eccentricity means that the most distant vertex in the network is a long way away, and a low eccentricity means that the most distant vertex is actually quite close [30]. The eccentricity of a node v in graph G is defined as: ( C_E(v) = \max_{t \in G} d_C(v, t) ), where ( d_C(v, t) ) is the minimum distance between v and t.</td>
<td>A central developer should have lower eccentricity as they will have direct communication ties with many other developers.</td>
</tr>
</tbody>
</table>

![Table 2: Core-Periphery partitioning of the projects](image)

<table>
<thead>
<tr>
<th>Project</th>
<th>Domain</th>
<th>Technology</th>
<th>Using Gerrit since</th>
<th>Requests mined*</th>
<th>Total # of developers</th>
<th>Size of the Core</th>
<th>Size of the pre-</th>
<th>% of core members</th>
<th>% of code commits by the core</th>
<th>% of the code reviews by the core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chromium OS</td>
<td>Operating System</td>
<td>C, C++</td>
<td>February, 2011</td>
<td>49,833</td>
<td>642</td>
<td>79</td>
<td>333</td>
<td>12.9%</td>
<td>64.7%</td>
<td>72.5%</td>
</tr>
<tr>
<td>ITK / VTK</td>
<td>Visualization Toolkit</td>
<td>C++</td>
<td>August, 2010</td>
<td>13,207</td>
<td>244</td>
<td>19</td>
<td>225</td>
<td>7.8%</td>
<td>57.0%</td>
<td>77.2%</td>
</tr>
<tr>
<td>LibreOffice</td>
<td>Office Application Suite</td>
<td>C++</td>
<td>March, 2012</td>
<td>6,347</td>
<td>207</td>
<td>20</td>
<td>187</td>
<td>9.7%</td>
<td>37.6%</td>
<td>88.0%</td>
</tr>
<tr>
<td>OmapiZoom</td>
<td>Mobile Development Platform</td>
<td>C</td>
<td>February, 2009</td>
<td>32,930</td>
<td>642</td>
<td>34</td>
<td>608</td>
<td>5.1%</td>
<td>34.3%</td>
<td>60.2%</td>
</tr>
<tr>
<td>OpenStack</td>
<td>Cloud Computing Software</td>
<td>Python, JavaScript</td>
<td>July, 2011</td>
<td>59,125</td>
<td>1,880</td>
<td>128</td>
<td>1,752</td>
<td>6.9%</td>
<td>53.6%</td>
<td>66.0%</td>
</tr>
<tr>
<td>OVIrt</td>
<td>Virtual Machine management</td>
<td>Java</td>
<td>October, 2011</td>
<td>21,316</td>
<td>193</td>
<td>20</td>
<td>173</td>
<td>10.4%</td>
<td>51.3%</td>
<td>61.1%</td>
</tr>
<tr>
<td>Qt Project</td>
<td>UI Framework</td>
<td>C, C++</td>
<td>May, 2011</td>
<td>71,732</td>
<td>888</td>
<td>63</td>
<td>825</td>
<td>7.1%</td>
<td>55.9%</td>
<td>66.1%</td>
</tr>
<tr>
<td>Typo3</td>
<td>Content Management System</td>
<td>PHP, JavaScript</td>
<td>August, 2010</td>
<td>24,374</td>
<td>387</td>
<td>30</td>
<td>357</td>
<td>7.8%</td>
<td>56.3%</td>
<td>71.0%</td>
</tr>
</tbody>
</table>

*Mined during January, 2014
able core-periphery relationship (Figure 1) [7]. CIK partitioned the 10 nodes into four core nodes and six peripheral nodes exactly the same as the Borgatti-Everett approach. Second, we used the single-core LibreOffice (left side of Figure 2). The two algorithms agreed on the characterization of 201 out of the 207 nodes (a 97% agreement). CIK identified 14 of the 17 core nodes identified by Borgatti-Everett, plus 6 additional nodes. Considering Borgatti and Everett’s algorithm as a gold standard, CIK has 70% precision, 82% recall, and 96% accuracy for the LibreOffice network.

Visual Inspection
Second we performed a visual inspection of the results from our new algorithm to ensure there were no obvious errors. In Figure 2 the core nodes are red and periphery nodes are gray. The visual inspection suggests that core nodes have the following three characteristics that would be expected of core nodes: 1) high degrees (i.e. larger size), 2) high weight edges (i.e. wide edges), and 3) physically central in the network.

Evaluation of Percentage of Core Members
Third, we evaluated whether the percentage of core members detected by our approach is similar to the percentage of core members reported in prior studies of OSS community structure. Using CIK, the percentage of core members across the eight projects was between 5% and 13% (Table 2). Previous work reported the percentage of core members as: 3.9% in the Apache project [28], 4.5% in the Linux kernel [23], 10%–20% for other OSS projects [36], 13% in FreeBSD [12], and between 18%–25% in seven Mozilla modules [28]. Furthermore, Dinh-Trong et al. hypothesized that, in OSS communities, the percentage of core members should be less than 25% [12]. Therefore, the percentage of core members resulting from our approach are consistent with previous OSS studies.

Evaluation of Percentage of Code Contributed by Core
Finally, we evaluated whether the percentage of the total code commits made by the core developers identified using CIK was consistent with the percentages reported prior OSS studies. CIK calculated the percentage of code committed by core members to be between 33% and 66% (Table 2). Previous work has reported this ratio to be: 19% in the Linux kernel [23], 55% in the PostgreSQL project [21], around 80% in the Apache project and FreeBSD [12, 28], and between 40%–50% in 19 other OSS projects [36]. The results from our approach are in line with these results. Lee and Cole’s Linux-kernel study is the only study that reported code review contributions from the core developers. They found core developers having more contribution share in the code review (32%) compared to code commits (19%) [23]. Consistent with this result, we found the review contributions from the core developers were higher in all eight projects (Table 2).

Finally, we used a different type of dataset than prior OSS studies (i.e. code review instead of code commit) and a different approach to identify the core developers, yet arrived at comparable results. This outcome provides more confidence in the correctness of the CIK approach.

Evaluation Summary
Based on evaluation, we can conclude that the CIK approach has 1) very high accuracy, 2) high precision/recall, 3) creates core nodes that pass the ‘eye-test’, 4) percentage of core members are consistent with prior OSS studies, and 5) percentage of code committed by core members are consistent with prior OSS studies. Therefore, we conclude that our novel CIK approach of categorizing an OSS network into core and periphery produces satisfactory partitions.

5. DATA ANALYSIS AND RESULTS

The following subsections describe the process we used to analyze the differences between the core and peripheral developers (Section 5.1) and then the results for the four hypotheses.

5.1 Analysis Approach
In our prior study, we observed that most review requests receive their first feedback within 2-4 hours, with some outliers taking up to a week. Similarly, most code review requests had a review interval of couple of days, with a few outliers taking much longer [8]. In the current study, we observed similar trends. For each developer we calculated the median first feedback interval, median review interval, acceptance rate, average number of patchsets per request, and average number of comments per review request. To avoid bias in the skewed distributions for first feedback interval and review interval (i.e. long tails), we used median for the central tendency. Because, the Shapiro-Wilk’s normality test indicated that the distributions of all metrics significantly differed from a normal distribution, we used non-parametric hypothesis tests (Mann-Whitney U) for each of the hypotheses introduced in Section 3.

5.2 H1: First Feedback Interval

Table 3 and Figure 3 show the median first feedback interval for the core and peripheral developers of each project. The first feedback interval is significantly lower for the core developers. Therefore, our results support H1. Additionally, the ‘Ratio’ column under H1 in Table 3 shows that the peripheral developers waited 1.8 to 6 times (or 1 to 12 hours) more for the first feedback on their code review requests.
Table 3: Statistical Results from Hypothesis Tests

<table>
<thead>
<tr>
<th>Project</th>
<th>H1: First Feedback Interval (hours)</th>
<th>H2: Review Interval (hours)</th>
<th>H3: Acceptance Rate</th>
<th>H4: No. of Patchsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chromium OS</td>
<td>Core 1.02 Periphery 2.78 Ratio 2.73 p&lt;.001*</td>
<td>Core 15.59 Periphery 27.47 Ratio 2.02 p&lt;.001*</td>
<td>Core 86.9% Periphery 44.1% Ratio p&lt;.001*</td>
<td>Core 1.30 Periphery 2.58 Ratio p=1.47</td>
</tr>
<tr>
<td>ITK/VTK</td>
<td>Core 3.47 Periphery 5.36 Ratio 1.55 p&lt;.001*</td>
<td>Core 22.62 Periphery 95.40 Ratio 4.22 p&lt;.001*</td>
<td>Core 80.8% Periphery 32.7% Ratio p&lt;.001*</td>
<td>Core 1.83 Periphery 2.19 Ratio p=2.65</td>
</tr>
<tr>
<td>LibreOffice</td>
<td>Core 6.36 Periphery 7.77 Ratio 3.69 p&lt;.001*</td>
<td>Core 11.37 Periphery 39.25 Ratio 3.45 p&lt;.001*</td>
<td>Core 90.4% Periphery 48.9% Ratio p&lt;.001*</td>
<td>Core 2.16 Periphery 2.41 Ratio p=0.02*</td>
</tr>
<tr>
<td>OpenStack</td>
<td>Core 0.64 Periphery 1.72 Ratio 2.69 p=0.01*</td>
<td>Core 73.59 Periphery 211.62 Ratio 2.7 p&lt;.001*</td>
<td>Core 70.1% Periphery 41.2% Ratio p&lt;.001*</td>
<td>Core 2.27 Periphery 2.28 Ratio p=1.30</td>
</tr>
<tr>
<td>OVirt</td>
<td>Core 2.47 Periphery 4.44 Ratio 1.8 p&lt;.001*</td>
<td>Core 56.52 Periphery 142.21 Ratio 2.52 p&lt;.001*</td>
<td>Core 84.0% Periphery 39.0% Ratio p&lt;.001*</td>
<td>Core 2.93 Periphery 3.07 Ratio p=0.016*</td>
</tr>
<tr>
<td>Qt Project</td>
<td>Core 5.56 Periphery 11.01 Ratio 1.98 p&lt;.001*</td>
<td>Core 41.38 Periphery 148.71 Ratio 3.59 p&lt;.001*</td>
<td>Core 86.6% Periphery 54.2% Ratio p&lt;.001*</td>
<td>Core 3.60 Periphery 3.90 Ratio p=0.872</td>
</tr>
<tr>
<td>Typo3</td>
<td>Core 0.04 Periphery 0.16 Ratio 4 p=0.003*</td>
<td>Core 3.99 Periphery 75.83 Ratio 19 p&lt;.001*</td>
<td>Core 91.3% Periphery 41.0% Ratio p&lt;.001*</td>
<td>Core 2.22 Periphery 2.64 Ratio p=1.143</td>
</tr>
</tbody>
</table>

* Indicates a statistically significant difference

5.3 H2: Review Interval

Table 3 and Figure 5 show the median review intervals for the core and peripheral developers on each project. The median review interval is significantly lower for the core developers on all projects. Therefore, our results support H2. Additionally, the ‘Ratio’ column under H2 in Table 3 shows that the peripheral developers waited 2 to 19 times (or 12 to 96 hours) longer than the core developers for the review process to complete.

Similar to H1, Figure 6 shows a representative example scatter-plot (for the Chromium project) comparing experience (measured by number of review requests posted) to median Review Interval. Again, the downward trendline indicates that median review interval is shorter for people who have posted more requests. Similar to H1, most of the peripheral developers that have posted less than 10 requests have much longer first feedback intervals (100 to 1000 hours). Therefore, peripheral newcomers with little or no recognition may have to wait longer for the first feedback of their posted review requests than someone who has more experience in the community.

5.4 H3: Acceptance Rate

Table 3 and Figure 7 show the average acceptance rate for the core and peripheral developers on each project. The acceptance rate is significantly higher for the core developers on all projects. Therefore, our results support H3.

Figure 5: Review Intervals

Figure 6: Experience vs. Review interval (Chromium OS)

Figure 7: Acceptance rate

Figure 8: Experience vs. Acceptance Rate (Typo3)
Similar to H1 and H2, Figure 8 shows an example scatter-plot (for Typo3) comparing experience (measured by number of review requests) and acceptance rate. The upward trendline indicates that acceptance rates are higher for developers with more project experience. The developers who have posted less than 10 review requests have the lowest acceptance rates. Therefore, peripheral developers, especially those who have posted less than 10 prior review requests have a lower probability of having their changes accepted.

### 5.5 H4: Number of Patches per Review Request

Table 3 and Figure 9 show the average number of patches required to get a change accepted for the core and peripheral developers on each project. For each project, the core developers required fewer patchsets than the peripheral developers, although the result is significant for only two projects. Therefore, the result for H4 is inconclusive overall. The differences among the core developers and peripheral developers are not as strong for this metric as they are for the metrics analyzed in H1-H3.

### 6. DISCUSSION

The results indicate that the core developers receive their first feedback more quickly and get a higher percentage of their changes accepted than the peripheral developers. Even though the core developers do not require fewer patchset submissions than the peripheral developers, they are able to complete review process more quickly. Taken together, these results suggest that core developers may also receive quicker subsequent feedback, although we did not test this hypothesis. This result is not surprising, core developers should enjoy some benefits, like quicker feedback, because they have built a reputation over time. However, if the feedback intervals are 2 to 19 times (or 12 to 96 hours) longer for the peripheral developers than for the core developers, those peripheral developers may lose the motivation to participate in the project. Conversely, if those peripheral developers received quicker feedback, they may be more motivated to contribute to the project. Because the vast majority of OSS participants are peripheral developers (> 80%), any measure that could increase their participation has the potential to have a great impact on the project.

The results also showed that, across all 8 projects, those developers at the bottom end of the experience spectrum, i.e. those that have posted less than 10 review requests, have very long first feedback and review intervals and very low acceptance rates. Generally, these developers are newcomers to the project, some of whom may intend to become regular contributors. Based on the results, we could hypothesize that as a contributor gains project experience and recognition by posting more review requests, s/he should begin to receive feedback on their changes more quickly. Because our results only show correlation and not causation, we must conduct further analysis to determine whether causation is present.

To gather some evidence for causation, we identified participants who started as newcomers and moved into the core through making a large number of contributions. Due to the wide variation in review intervals across projects, making cross-project comparisons would be difficult. We chose to focus our analysis on the Qt project because it had the most review requests of any project in this study. We identified 63 core developer in Qt as of January 2014. Of those, 59 developers belonged to the core during March 2012 (using 10 months’ code review data). Only four developers first began contributing to Qt after March 2012 and moved into the core through their contributions. Figure 10 shows the median first feedback intervals, median first review intervals and average acceptance rate for those four developers over project experience. All four developers had a long first feedback and review interval at the beginning. After posting 10-25 review requests, their first feedback and review intervals were more in line with the other core developers. This trend seems to provide evidence for the conclusion that as developer gain reputation and experience (i.e. by posting more review requests) their requests receive quicker feedback and they complete the review process quicker.

Conversely, when examining acceptance rate, these developers do not fit the expected pattern established by the results of H3. Three of the four developers had an acceptance rate greater than 80% for their first 10 review requests. However, most of the prospective OSS joiners are not successful [13], and even may not be able to get past 10 review submissions. Together, these two points suggest that a high initial acceptance rate may be an indicator that a newcomer has the potential to move into the core. More analysis is required to draw any conclusion about this observation.

Based on the results of this study, we hypothesize that OSS projects should benefit by developing a triage process to help newcomers receive quicker feedback. Even though it may not be possible to increase the acceptance rate for newcomers (because it could compromise project quality), OSS projects can encourage newcomers by providing quicker feedback on review requests. If a newcomer has to wait too long for feedback, s/he may feel ignored, lose interest, and ultimately leave the project. This attrition may have a negative effect on an OSS project. Research suggests that projects that fail to attract and, more importantly, to retain new contributors cannot survive [13]. Conversely, if a newcomer receives feedback quickly, s/he is more likely to make the requested changes so his/her code can be merged into the project codebase. Having his/her changes accepted will help a newcomer become recognized and build reputation within the project [33]. This recognition will, in turn, motivate him/her to contribute more [19] and increase the overall project output.

### 7. THREATS TO VALIDITY

This section is organized around four common threats to validity.

#### 7.1 Internal validity

The primary internal validity threat is project selection. We only included projects that use modern code review supported by Gerrit, we included the majority of the publicly accessible Gerrit-based projects that contain a large number of code review requests. Furthermore, it is possible that projects supported by other code review tools could have behaved differently. We think this threat is minimal for three reasons: 1) all code review tools support the same basic purpose, i.e. detecting defects and improving the code, 2) the basic workflow (i.e. authors posting code, reviewers commenting about code snippets, and code requiring approval from reviewer
before integration) of most of the code review tools are similar 3) we did not use any Gerrit-specific feature/attributes in this study. Therefore, we believe the project selection threat is minimal.

7.2 Construct validity
The primary threat to construct validity is regarding our approach to partition an OSS community into core and periphery groups. It is possible that our approach placed some developers in the wrong group. However, the validation (Section 4.4) increases our confidence that the number misgrouped developers is small. Because each project had a large number of developers, a few misgrouped developers will not significantly alter the results.

Second, we assume that the code review outcomes measured in this study (i.e. first feedback time, review interval, acceptance rate, and number of patchsets per review request) are influenced by a developer’s position in the code review social network. However, there may be number of unmeasured confounding factors that could influence a review outcome (e.g., availability of developers, code complexity, status of the author as a volunteer or a paid contributor, and licensing terms). There is no evidence that our partitioning approach or our measurements would be systematically biased based on any of these confounding factors. Furthermore, the fact that the results were similar across all eight projects provides additional confidence that any confounding factors did not significantly impact the study results.

Finally, the assumption that core developers have more recognition than the peripheral developers is a more minor threat. While this assumption is likely true for most core developers, there may be few exceptions. As an extreme example, if an iconic OSS figure (e.g. Linus Torvalds, or Guido Van Rossum) or another well-known developer makes occasional contributions to an OSS project, he would be classified as a peripheral developer but would have more recognition in the community than the many of the core members. Because the projects used in this study had a large number of developers, any exceptions like this would be minimal and would not impact the overall results.

7.3 External validity
The OSS projects in this study vary across domains, languages, age, and governance. In addition, we analyzed a large number of code review requests for each project. Therefore, we believe these results can be generalized to many other OSS projects. However, because of the wide variety of OSS project characteristics, we do not claim that these results are universal for all OSS projects. Drawing a more general conclusion would require a family of experiments [2] that included OSS projects of all types. To encourage a type of replication, we have made our scripts available to other interested researchers.

7.4 Conclusion validity
The use of large dataset drawn from eight projects, which produced similar result across those eight different projects, boosts confidence about the validity of the results. We tested all data for normality prior to conducting statistical analyses and used appropriate tests based upon the results of the normality test. We used popular and well-known social network analysis tools for this study. Therefore, threats to conclusion validity are minimal.

8. CONCLUSION
To identify the impact of an OSS developer’s reputation on the outcome of his/her contributions, we performed a social network analysis of the code review data from eight popular OSS projects based on the assumption that core developers have a better reputation within the community than the peripheral developers. The results suggest that core developers indeed enjoy quicker first feedback intervals, shorter review intervals, and higher code acceptance rates than the peripheral developers. The lack of an established reputation may result in peripheral developers waiting 2 to 19 times (or 12 to 96 hours) longer than core developers to complete the review process.

Therefore, the first key contribution of this study is the empirical evidence regarding the effect of OSS developers’ reputation on code review outcome. Furthermore, we can conclude that a developer’s reputation has the most impact when s/he is a newcomer. However, once s/he has built some recognition by posting a minimum number of review requests, which we found to be between 10-25, s/he may enjoy shorter feedback intervals, which may be similar to the core developers. Since a delay in receiving feedback on his/her review requests may negatively motivate a newcomer’s continued participation in the project, OSS communities should have means (e.g. review triage, or reviewer recommendation systems) in place to ensure timely feedback for newcomers review requests.

Another key contribution of this study is our novel “Core Identification using K-means (CIK)” approach. There has not been any existing approach to identify core nodes in large multi-core networks (e.g., most of the popular OSS communities). We believe our approach will be useful for other researches to study the core nodes in large multi-core networks.

Acknowledgments
This research is partially supported by the US National Science Foundation Grant No. 1322276.

9. REFERENCES


