Identifying Features in Forks

Shurui Zhou
Carnegie Mellon University

Ştefan Stănciulescu
IT University of Copenhagen

Olaf Leßenich
University of Passau

Yingfei Xiong
Peking University

Andrzej Wąsowski
IT University of Copenhagen

Christian Kästner
Carnegie Mellon University

ABSTRACT

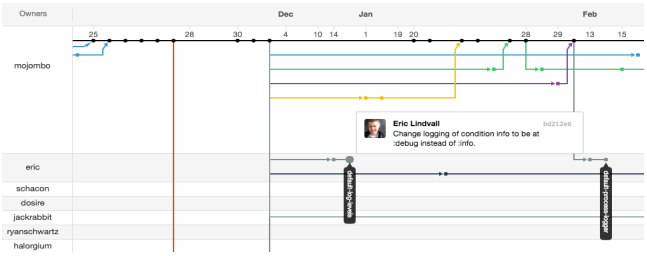
Fork-based development has been widely used both in open source communities and in industry, because it gives developers flexibility to modify their own fork without affecting others. Unfortunately, this mechanism has downsides: When the number of forks becomes large, it is difficult for developers to get or maintain an overview of activities in the forks. Current tools provide little help. We introduce Infox, an approach to automatically identify non-merged features in forks and to generate an overview of active forks in a project. The approach clusters cohesive code fragments using code and network-analysis techniques and uses information-retrieval techniques to label clusters with keywords. The clustering is effective, with 90 % accuracy on a set of known features. In addition, a human-subject evaluation shows that Infox can provide actionable insight for developers of forks.

ACM Reference Format:
Shurui Zhou, Ştefan Stănciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wąsowski, and Christian Kästner. 2018. Identifying Features in Forks. In *ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180205

1 INTRODUCTION

Forking is a lightweight and easy mechanism that allows developers, both in open source and in industry, to start development from an existing codebase, while having the freedom and independence to make any necessary modifications [8, 23, 29, 75]. Forking was always available by simply copying code files, and version control systems have long supported branches, but recent advances in distributed version control systems (e.g., 'git clone') and social coding platforms (e.g., GitHub fork) have made *fork-based development* relatively easy and popular by providing support for tracking changes across multiple repositories with a common vocabulary and mechanism for integrating changes back [19].

While easy to use and popular in practice, fork-based development has well known downsides. When developers each create their own fork and develop independently, their contributions are usually not easily visible to others, unless they make an active attempt in merging their changes back into the original project. When the number of forks grows, it becomes very difficult to keep track of decentralized development activity in many forks (as we will illustrate in Sec. 2). The key problem is that it is *difficult to maintain an overview* of what happens in individual forks and thus of the project's scope and direction. For fork-based development in industrial contexts, both Berger et al. and Duc et al. found that it is hard for individual teams to know who is doing what, which features exist elsewhere, and what code changes are made in other forks [6, 24]. Several open-source developers that we interviewed for this paper indicated that they are interested in what happens in other forks, but cannot effectively explore them with current technology, such as GitHub's network graph shown in Fig. 1a: *"I care, but, it is very hard to track all of the forks."* This developer is using SourceTree, which visualizes commit history of a repository through GUI, to explore code changes in other forks one by one, and he said *"it is just difficult"* [P5]; *"I do not have much visibility of the forks. They are too many, and it is overwhelming to keep track of them"* [P9]. The difficulty to maintain an overview of forks leads to several additional problems, such as redundant development, lost contributions and suboptimal forking point, as we will discuss in Section 2.



(a) GitHub network graph

Smoothieware / Smoothieware
Watch 179 Star 530 Fork 573
#Active forks (within a year): 89
#Forks have un-submitted code changes: 33

Cluster	OpenBuilds/Smoothieware, last commmit: 16 days ago	LOC
com.	[com, substr, smoothie, setcursor, build, lcd, c_str, openbuilds]	46
lcd.	[lcdwidth, shift, glcd_font, fb_size, framebuffer, memory, is_sh1106, size_sh1106, data, framebuffe	10
con.	[contrast, apex, is_sh1106, support, contrast, reversed, oled, ol, sh1106]	5
ol_c.	[ol_checksum, new, variant, checksum, sh1106_ol, cksm, lcd_cksm, support, lcd, st7565]	2

Cluster	lciscon/Smoothieware, last commmit: Jan 27	LOC
bed,.	[bed, gcode, div, ha_letter, home_offset, gcode_receiv, nullstream, number, correct_checksum, st	28

Cluster	arhi/Smoothieware, last commmit: Jan 3	LOC
amp.	[ampmod1_pin, adc_valu, ampmod2_pin, thekernel, value, by_default, name_checksum, as_numbe	92
add.	[added, example, config, function, pt100, class, streamoutput, snippet]	1

(b) Infox overview

Figure 1: GitHub's network graph shows commits across known forks, but is difficult to use to gain an overview of activities in projects with many forks. Infox's overview summarizes features in active forks.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180205

The goal of our work is to identify and label cohesive code changes, *features*, among changes in forks to provide a compact overview of features and their implementations. This is a step to establish an overview of development activities in various forks of a project.

In contrast to GitHub's network view (Fig. 1a), we deemphasize commits, which frequently have unreliable descriptions and frequently are unreliable indicators of cohesive functionality, as it is common that commits tangle code of multiple features and even more common that a single feature is scattered across multiple commits [5, 38, 39, 43, 48, 54]. Instead, we *cluster changed code* based on relationships and dependencies within those code fragments and label each feature with *representative keywords* extracted from commit messages, code, and comments. Technically, we take inspiration from ClusterChanges [5] to untangle code changes during code review based on a graph of code dependencies and repurpose the idea for our problem; furthermore we incorporate community-detection techniques [34] to refine an initial clustering and information-retrieval techniques [67] for deriving concise labels (See Fig. 1b).

We implemented our analysis in the Infox tool[1] that produces web pages describing the features in individual forks and across multiple forks of a C/C++ project as illustrated in Fig. 1. In a mixed-methods evaluation, we demonstrate (1) that Infox is effective at identifying features among changes in 10 open-source projects with a median accuracy of 90 percent, (2) that Infox's technical innovations improve clustering accuracy over an existing state-of-the-art technique designed to cluster individual commits [5], and (3) that the produced overview provides useful insights for contributors and maintainers of projects with many forks.

To summarize, we contribute (a) Infox, an approach and corresponding tool, which automatically identifies and summarizes features in forks of a project, using source code analysis, community detection, and information-retrieval techniques, and (b) evidence that Infox improves accuracy over existing techniques and provides meaningful insights to maintainers of forks.

While Infox currently aims at supporting exploration and navigation by summarizing features, it lays a foundation for future interactive tool support that can refine and persist features (e.g., for a product-line platform [4, 9, 65]) and support developers in merging selective changes across forks (e.g., generating pull requests).

2 MONITORING FORKS IN PRACTICE

GitHub's main facility to navigate forks is the network view (Fig. 1), which visualizes the history of commits over time across all branches and forks of a project. This cross-fork visualization provides transparency to developers who want to track ongoing changes by others, want to know who is active and what they are trying to do with the code [19]. For example, one of the developers we have interviewed said: *"I check the more updated forks. I think this view is helpful, because I am not gonna look at all 60 forks. 60 is a lot, probably this project has thousands, that will be ridiculous. I will never do that"* [P4].

Although the network view is a good starting point to understand how the project evolves, it is tedious and time consuming to use if a project has many forks. In order to see older history, users click and drag within the graph, and if users want to see the commit information, they hover the mouse over each commit dot and read the commit message. Also, they *"have to scroll back a lot to find the fork point and then go to the end again for seeing what changed since then in the parent and in the fork"* [2]. If developers want to investigate the code changes of certain forks, they have to manually open and check each fork. As one developer stated *"I don't look at the graphs on GitHub… it is very hard to find the data, you have to scroll for 5 minutes to find stuff"* [P5]. The view does not even load when there are over 1000 forks, no matter they are active or inactive.

Subsequently, it is difficult for developers to maintain an overview of forks, which can lead to several additional problems:

- *Redundant development:* Unaware of activities in other forks, developers may reimplement functionality already developed elsewhere. Stănciulescu et al. report that, in an open source project, 14 percent of all pull requests were rejected because of concurrent development [71]. Redundant development further leads to merge conflicts, which would demotivate or prevent developers from continuously contributing to the repository [35, 69], and significantly increases the maintenance effort for maintainers [23, 71]. A developer we interviewed for this paper described the problem as follows: *"I think there are a lot of people who have done work twice, and coded in completely different coding style"* [P3].
- *Lost contributions:* Developers may explore interesting ideas, fix bugs, or add useful features in forks, but unless they contribute those changes back to the original project, those contributions are easily lost to the larger community. Even though contemporary social-coding platforms list all known forks (see Fig. 1), project maintainers are unlikely to identify interesting contributions among the thousands of forks many open source projects attract. Furthermore, even when a feature of interest is identified in a fork, because of independent development in each fork, it can be difficult to port features from one fork to another [23]. A frequently mentioned complaint is that forks rarely change the Readme file to describe what the fork changes.
- *Suboptimal forking point:* Without an overview of forks and their different contributions, developers might not fork from the codebase that is closest to their intended goals. Dubinsky et al. report that in industrial fork-based development projects developers often struggle to identify which of multiple existing forks to select as a starting point [23].

There are many different reasons to fork a project: adding a feature, fixing a bug, preparing a pull request, continuing an abandoned project, customizing or configuring the project to create a variant, or making a private copy [23, 52, 64, 71]. In fact, many forks of a project tend to be inactive. For example, one of the subject systems in our study, *Smoothieware*, has 623 forks in total, of which 89 forks performed unique non-merged code changes, of which 33 were active within the last 12 months. To an observer, the function of a fork and its activity level is difficult to identify; somebody looking for interesting activities (e.g., forks developing features, fixing bugs or experimenting with code [71]) will often has to navigate all forks.

[1] Infox is short for **I**de**N**tifying **F**eatures in f**Or**K**S**.
Source code is publicly available at https://github.com/shuiblue/INFOX.
A lightweight web service is available at: forks-insight.com.

GitHub has not addressed the increasing issue with navigating many forks despite many feature requests [1, 2]. External developers have explored only very basic improvements, most notably a web browser extension that shows the most starred fork, for the common use case of identifying an active fork to an abandoned project [3]. In our user study, we learned there are mainly two ways practitioner currently use to find interesting forks: Developers either look at forks they know about and go through the fork's commits and commit messages (e.g., *"I do compare one branch to another"* [P5]), or use Google and GitHub to search for particular keywords. Searching is mentioned by several participants as the preferred choice, e.g., *"I usually will use Google but set to only look inside GitHub"* [P6].

In this work, we suggest a more systematic approach to create an overview of forks that identifies the changes in each forks, clusters them into features, and provides concise descriptions through a set of characteristic keywords.

3 INFOX

INFOX identifies and labels features within a larger change of a fork. It takes the diff between the latest commit of the upstream (source snapshot) and the latest commit of the fork (target snapshot) from GitHub, which returns the non-merged changes from fork.[2] Then it proceeds in three steps (as shown in Fig. 2):

- Identify a dependency graph among all added or changed lines of code by parsing and analyzing the code for multiple kinds of dependencies (Sec. 3.1).
- Cluster the lines of the change based on the dependency graph using a community-detection technique, mapping each line of code to a feature, such that lines with many connections in the dependency graph are mapped to the same feature (Sec. 3.2).
- Label each cluster by extracting representative keywords with an information-retrieval technique (Sec. 3.3).

The first step is inspired by ClusterChanges, an approach to untangle code in commits for code review [5]. ClusterChanges clusters changed code fragments based on a dependency graph of lines of code. We adopt this idea for a different purpose (identifying and naming features in multiple forks rather than untangling changes in a single commit) and we extend the approach with additional kinds of dependency edges, additional steps in the clustering process, and labeling of clusters, as we will explain.

3.1 Generating a dependency graph

We generate a dependency graph for all lines of code of the target snapshot by parsing the target snapshot and analyzing the resulting abstract syntax tree. We add edges between lines for several kinds of relationships of code fragments within those lines that may indicate that the two fragments are that are more likely to be related. We collect the following kinds of dependencies, which we also illustrate on a simple excerpt of an email system in Fig. 3:

- *Definition-usage edges:* We add edges between the definition and use of functions and variables in the program, and the definition and use of structs or classes and their members. We conjecture that def-use relationships between two code fragments often point to two code fragments that fulfill a joint purpose and are thus more likely to be part of the same cohesive change in a larger change.
- *Control-flow edges:* We generate a control-flow graph for the source code and add edges between two lines if there is a control dependency relation between the statements of each line. In line with Emerson's cohesion metrics [27], we think that the flow of control information contributes to the cohesion of code changes.
- *Adjacency and hierarchy edges:* We add edges between consecutive lines and lines that represent hierarchical structures in the source code (struct/class members point to the outer struct definition). Adjacency edges and hierarchy edges represent the structure of the source code and indicate that code fragments that are located close to each other are more likely to belong to the same cohesive fragment than code fragments scattered across different places.

The result is a labeled, weighted, undirected graph, in which nodes represent lines of code and edges represent the identified dependencies listed above. We assign a low weight of 1 for adjacency edges, and a weight of 5 for all other edges. Intuitively, semantic dependencies in the program should be stronger indicators of features than structural relations. We use an undirected dependency graph, as our experiments showed no benefit in maintaining directionality.

Using a *diff* command between the source and the target snapshot, we identify and mark all nodes that have been added or changed in the target snapshot (highlighted in Fig. 2).

Compared to ClusterChanges [5], we add edges between nodes with hierarchical and control-flow relations, and add weights.

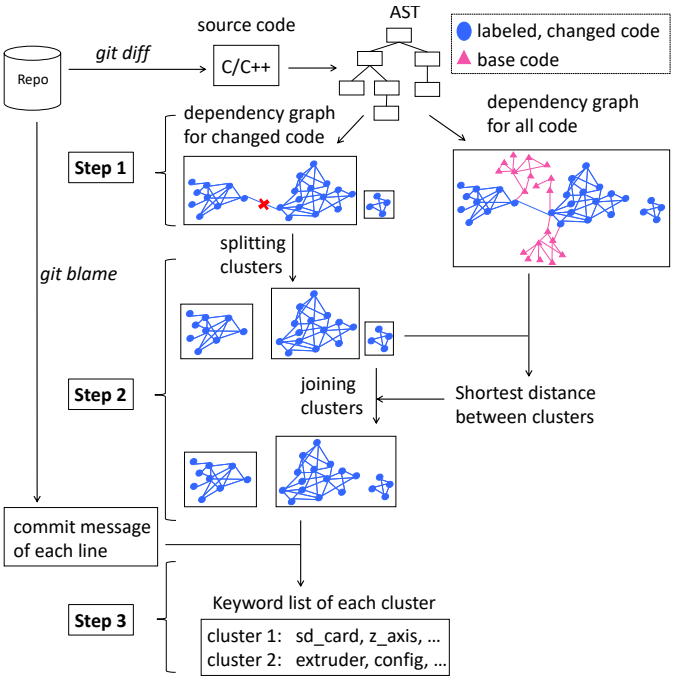


Figure 2: Generating and clustering dependency graphs to identify features, and labeling features.

[2]While developed for changes in a single fork, our approach can be technically used to cluster the changes between any two code snapshots, including two commits in a single repository or two copies of code maintained without a version control system.

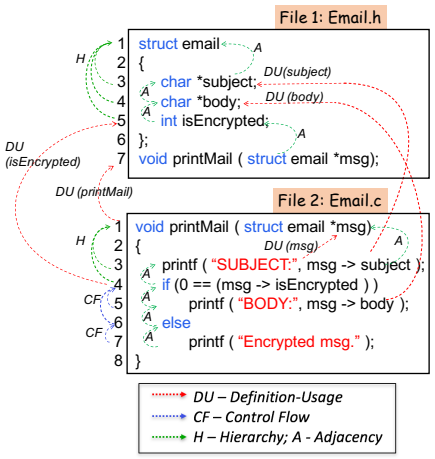


Figure 3: Edge examples of an email system.

3.2 Identifying features by clustering the graph

Given the labeled, weighted, undirected dependency graph of the target snapshot, we identify clusters of nodes that have many edges within the cluster but few edges across clusters (known as *community detection* in the network analysis community [32, 74]). We interpret each cluster and the corresponding lines of code as a feature. We start with an initial simple clustering step, but provide additional means to further split and join clusters that can be used in an interactive tool or applied automatically using heuristics.

Initial clusters. In line with ClusterChanges [5], we establish initial clusters by removing all unlabeled nodes (see Fig. 2) from the dependency graph (i.e., all nodes that have not been added or changed between source and target) and by detecting *connected components* in the resulting graph. Each connected component is considered as a feature.

This simple splitting works well for many changes, for example, grouping together adjacent code fragments and new function definitions with their corresponding calls. Unfortunately, for other changes, including large and tangled changes—that we see more frequently in forks than in individual commits—this initial clustering is susceptible to problems where two unrelated features are merged, just because their implementations share a single adjacency edge in one place in the source code. Similarly, some code fragments may belong together but have no link within the changed code, such as multiple scattered code fragments calling the same previously existing logging function. For that reason, we go beyond the work of ClusterChanges [5] and provide additional (optional) support for splitting and joining clusters further.

Splitting clusters. Community detection identifies modules in a graph according to the structural position of nodes [32]. In community detection, the key optimization criterion is to maintain more connections within the module than across modules. We use community detection to split large clusters into smaller ones that are only loosely connected.

We adopt a state-of-the-art community-detection algorithm by Girvan and Newman [34]. Its idea is to count the number of shortest paths between node pairs. This count, weighted by the edge weight, is called the *edge-betweenness* score. The edges with higher betweenness tend to be the bridge between communities. The clustering algorithm iteratively removes the edges with the highest *edge-betweenness* score from the original graph.

In the example of Fig. 2 (Step 1), the highlighted edge is the one with the highest betweenness score, bridging two otherwise highly interconnected clusters. Our algorithm removes this edge, splitting the large cluster into two smaller ones (Step 2).

Note that community detection has no natural stop criteria. The algorithm can continue until the last edge is removed, creating singleton clusters. In practice, several heuristic stop criteria exist, such as maximizing a modularity metric [34] or stopping when a given maximum number of cut edges do not yet result in a new cluster. Since, despite experiments, we could not identify a single robust stop criterion for our problem, we primarily envision splitting in an interactive setting, in which developers can request to split a large feature into two using the community-detection algorithm, if they judge this to be beneficial.

Joining clusters. Scattered implementations of a single conceptual feature may result in graph components without any connecting edge. In our experience, this sometimes generates sets of small clusters that appear to be highly related (e.g., call the same function, use the same variable), but have no dependency edge within the changed code. To identify these as a single feature nonetheless, we analyze how those clusters are related in the context of the entire implementation, not just the added or changed lines of code.

To this end, we compute the distance between two clusters in the entire dependency graph that includes the unlabled nodes representing unchanged lines of code that are the same in the source and target snapshot. Given two clusters, we compute the distance of two clusters as the shortest distance between any pair of nodes, in which each node belongs to one of each clusters. In our example, in Fig. 2, the two initial clusters are separated by only a single unmarked node, indicating that they might be joined.

Again, there is flexibility in selecting thresholds about when to join two clusters. In addition to interactive mechanisms, we apply joins by default for pairs of clusters that are separated by a single unlabled node when at least one of the clusters is smaller than 50 nodes (lines of code). We arrived at this default threshold after observing, across a large number of forks, that small clusters are more frequently affected by this, whereas large clusters are more likely to already share an edge.

3.3 Labeling features

After identifying features with clusters in the dependency graph of the changed code (possibly with additional splitting and joining), we can already show the clusters to developers. However, to allow them to gain an overview of a fork's changes quickly without having to read a large amount of source code, we label each feature with representative keywords.

In contrast to GitHub's network graph (see Fig. 1), which only shows individual commit messages, we compute representative keywords from multiple sources. We use commit messages in the process, but do not rely on them, because (1) commit messages are often too verbose to consume quickly, because (2) as discussed, commits do not always align with features, and because (3) we do not consider the text of commit messages as reliable descriptors.

```
1    if (dual_x_carriage == DXC_DUPLICATION){
2        setTarget(duplicate_extruder_temp);
3        duplicate_extruder_temp_offset = code_value();
4        duplicate_extruder_x_offset = max(i, t);
5        SERIAL_ECHO(extruder_offset[0]);
6        extruder_duplication_enabled = false;
7    }
```

Figure 4: Source code excerpt from Marlin.

Instead, we use information-retrieval techniques to identify keywords that are distinctive for a given feature, meaning that those keywords represent the feature better than other features or the base implementation. Specifically, we proceed in three steps:

- First, we collect a corpus of text for each feature and an additional corpus for the unmodified source code. For each feature, the corpus contains (verbatim) all lines of code associated with this feature, including variable names and function names. We also include source-code comments that may provide additional explanations. Comments are added to corresponding clusters based on their line number. Finally, we identify the commits that introduced the changed lines of a feature (using 'git blame') and add all corresponding commit messages to the corpus.
- Second, we tokenize each corpus (e.g., splitting variable names at underscores [10]) and perform the standard normalization techniques of stemming (e.g., unifying variations of words such as duplicate, duplicated, and duplication) and removing stop words (specifically reserved keywords such as *int, sizeof, switch, and struct*).
- Third, we identify keywords that are important in one corpus as compared to all other corpora using the well-known *Term Frequency Inverse Document Frequency (TF-IDF)* scoring technique [67]. The importance of a keyword (its TF-IDF score) increases proportionally to the number of times a word appears in the feature's corpus but is offset by the frequency of the word in the other feature's corpora [46]. We calculate the TF-IDF score of each word and of each 2-gram (unique sequence of two words [72]) in the feature's corpus. We report the five highest scoring words and five highest scoring 2-grams as labels for the feature.

For example, consider the code snippet from the Marlin 3D printer firmware[3] in Fig. 4, which we represent by the relevant keywords *duplicate_extruder*, *extruder_temp*, *offset*, *dual_x*, *x_carriage*, which are common in this code fragment but not elsewhere in the firmware implementation.

We arrived at our solution of tokenizing composed variable names (with underscore) and using 2-grams after some experimentation. On the one hand, composed variable names (e.g., *duplicate_extruder_x_offset* in Fig. 4) are often too specific and dominate the TF-IDF score, such that all keywords are long and often similar variable names. On the other hand, we do not want to discard them entirely as they often include descriptive parts that represent the feature. Finally, tokenizing all composed words sometimes leads to overly generic words, for example, unable to distinguish the different kinds of extruders in 3D printers. Tokenization combined with 2-grams provides a compromise that can pick up common combinations of words without relying too much on specific long combinations and generic short words.

[3] https://github.com/MarlinFirmware/Marlin

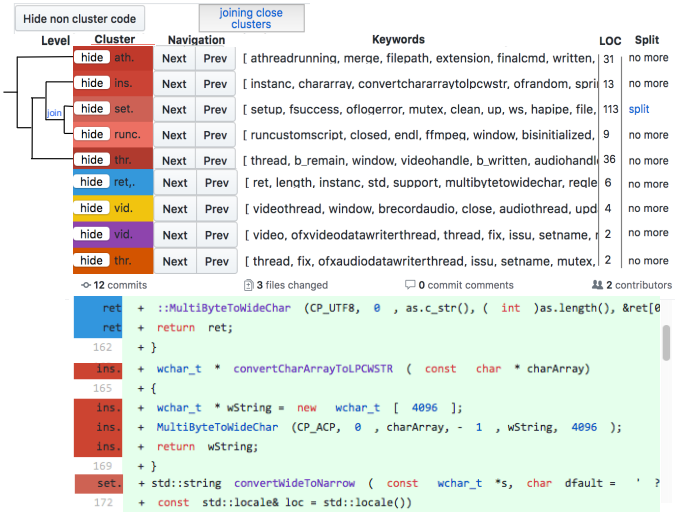


Figure 5: Features in fork *DomAmato/ofxVideoRecorder*; tree view displaying hierarchical relation between split features; colors related code to features.

4 IMPLEMENTATION & USER INTERFACE

We implemented Infox for C and C++ code in a tool of the same name. Infox takes a link to a GitHub project and collects all active forks. For each fork, it downloads the latest revision of each as target snapshot. Unless instructed otherwise, it takes the latest revision of upstream repository as that fork's source snapshot. As output, Infox produces an HTML file that contains summaries of features and keywords for all analyzed forks, ranked by the time of their last commits, as shown in Fig. 1b. In addition, for each fork, it produces an HTML file that maps the features to source code (using colors, similar to FeatureCommander [30]) as shown in Fig. 5. Navigation buttons allow to jump between scattered code fragments of a feature.

To build a dependency graph, Infox parses C/C++ code with srcML [14] and performs a lightweight name-resolution analysis to detect def-use edges. Since reliably identifying all such edges in a complex language as C++ is difficult, our implementation is unsound, but provides a fast and sufficient approximation for our experiments. For example, Infox does not disambiguate function pointers or other advanced language constructs.

Splitting and joining is currently implemented such that developers can interact with the web page and select which additional features to split and which to join. Splits are currently precomputed for features larger than 50 lines, as are joins for small features (by generating multiple static HTML pages through which the user navigates). This can easily be replaced by on-demand computations on a web server. Split clusters are illustrated with a hierarchy allowing users to track and undo splits.

5 EVALUATION

We evaluate Infox with regard to effectiveness and usability. Specifically, we address four research questions:

RQ1: To what extent do identified clusters correspond to features? We measure the effectiveness of Infox's clustering approach by comparing how well the clusters match previously labeled features in the code. To that end, we will establish a ground truth of features in multiple code bases. We further compare the result of Infox with ClusterChanges [5].

RQ2: What design decisions in Infox are significant to cluster cohesive code changes? We aim to understand the factors that influence the effectiveness of Infox. Specifically, we investigate how sensitive Infox is to different kinds of edges in the dependency graph and to the splitting and joining steps.

RQ3: To what extent do developers agree with Infox's clustering result? Complementing RQ1, we explore whether fork maintainers in open-source projects agree with how Infox divides and labels their own contributions.

RQ4: Can Infox help developers to gain a better overview of repository forks? We investigate whether Infox helps developers to gain new and useful information about a project's many forks, such as recognizing useful contributions or redundant development in other forks.

We answer the first two research questions in a *controlled setting*, in which we *quantitatively* measure the accuracy of different clustering strategies on a number of subject systems for which we establish some ground truth as benchmark. Subsequently, we *qualitatively* answer the remaining two research questions in a *human-subject study*, in which we discuss Infox's results with 11 developers of forks of popular open source systems. The studies are complementary, allowing us to both (a) systematically explore a large number of diverse scenarios while controlling several confounds and deliberately exploring the effects of changing independent variables (internal validity), as well as (b) validate in a practical setting how developers can benefit from the approach in their day-to-day development (external validity).

5.1 Quantitative Study (RQ1 & RQ2)

In a first study, we answer RQ1 and RQ2 in a controlled setting by quantitatively comparing clustering results of Infox and ClusterChanges against a ground truth of known features in a number of open-source projects.

Establishing ground truth. A key challenge in evaluating approaches that identify features and cohesive code fragments (including a vast amount of literature on the concept-location problem) is to establish ground truth—a reliable data set defining which code fragments belong to which features. Once such ground truth is established, it is easy to define an accuracy measure and to compare different approaches and their variants. There are many different ways to establish ground truth, each with their own advantages and disadvantages, including (1) asking researchers or practitioners to manually assign features to code fragments [25, 63] (possibly biased and subjective, possibly low inter-rater reliability, expensive), (2) using indirect indicators such as code committed in a single commit or by a single author [22, 77] (questionable reliability), (3) using results of other tools as reference [40, 41, 61] (questionable reliability), or (4) using existing traceability mappings created for compliance reasons [12, 13, 16–18, 22] (uncommon practice outside industrial safety-critical systems). In this paper, we use a new, different approach and use existing mappings of code fragments to features through #ifdef directives in C/C++ code.

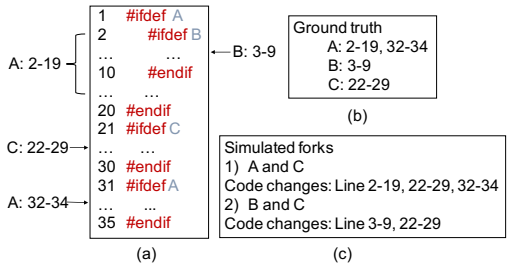


Figure 6: Extracting preprocessor-based ground truth and simulating forks.

The preprocessor is commonly used in C/C++ code to implement optional features and support portability such that users can customize their builds by instructing the preprocessors which features to include [28, 49, 51]. We argue that *#ifdef*-guarded code fragments are often good approximations of features (extensions, alternatives) that developers might add in a fork of a software system. In fact, in some systems like the Marlin 3D-printer firmware, developers often add *#ifdef* guards around code blocks that they integrate back into the upstream repository as pull requests [71].

Given a C/C++ project, we identify all preprocessor macros that correspond to features of the system, excluding macros that are used for low-level portability issues. For each macro, we identify all code fragments that are guarded by this macro. We consider this macro-to-code mapping as the ground truth for features in the experiments, as illustrated in Fig. 6. Extracting ground truth from preprocessor annotations has the advantage that those annotations have been added by practitioners independently of our experiments and that they can be extracted automatically at scale. As developers typically want to compile the code with and without those features, the mapping is typically well maintained as part of normal development activities and the features correspond to units of implementations that developers and users care about. Note that preprocessor annotations do not map all of the project's code to features, but this is not necessary, because we only want to cluster the code changed in forks. We describe next how we simulate such changes to forks.

Simulating forks. For a given project, we simulate multiple forks, of which each adds multiple features. To that end, we select a subset of features in the project and create the *source snapshot* by removing all code corresponding to those features (based on the ground-truth mapping), whereas we remove only the #ifdef directives but not the corresponding implementations from the *target snapshot*. That is, source and target snapshot differ exactly in the implementation of the selected features. We then evaluate whether Infox can cluster the changed code into the features originally defined by the project's developers. By selecting different sets of features, we can generate different simulated forks for the same project.

Since Infox divides all code added in a fork into non-overlapping clusters, we avoided nested macros in one feature combination when generating simulated forks. Technically, we select macros incrementally and randomly, and discard any macro for which code overlaps with previously selected macros until we found the desired

Table 1: Subject projects

Software System	Domain	LOC	#F	F-LOC
Cherokee	web server	51,878	328	7,679
clamav	anti-virus program	75,345	285	10,809
ghostscript	postscript interpreter	442,021	816	21,864
Marlin	3D printer firmware	190,799	280	26,395
MPSolve	mathematical moftware	10,181	17	1524
openvpn	security application	38,285	276	23,288
subversion	revision control system	509,337	409	28,443
tcl	program interpreter	135,183	2,481	26,618
xorg-server	X server	527,871	1,360	95,227
xterm	terminal emulator	49,621	453	19,208

LOC: lines of code; #F: number of features macros;
F-LOC: size of features in LOC

number of non-overlapping macros. For example, in Fig. 6, macro B is nested in macro A, thus we may generate simulated forks with A and C, and B and C, but not forks with A and B.

In order to evaluate the *effectiveness* and *robustness* of INFOX, we ran INFOX on multiple projects and on multiple simulated forks per project. We explored all combinations of the following three *experimental parameters* and generated 10 simulated forks for each combination, resulting in 156 simulated forks per project.

- *Number of macros:* We selected between 3 and 15 macros per simulated fork, simulating smaller and larger changes.
- *Proximity:* We either selected all macros (a) from the same file or (b) from different files, simulating more and less heavily tangled features.
- *Feature size:* We sorted all features by size (lines of code) and split them equally into smaller and larger features. We then sampled either (a) twice as many large features than small features or (b) twice as many small features than large features, thus further varying simulated forks.

Subject systems. We use open-source software systems implemented in C/C++ with #ifdef annotations. We selected projects differing in domain, size, and number of features from existing research corpora [49, 71]. Table 1 lists the 10 selected systems.

Accuracy (dependent variable). To evaluate how well a clustering result matches the ground truth, we use a standard accuracy metric from community detection [74]: Considering all possible pairs of nodes ($2n(n-1)$ pairs for n nodes), accuracy is the ratio of correctly clustered pairs (denoted as CCPs) among all the pairs of nodes ($accuracy = \frac{CCPs}{2n(n-1)}$). A pair is correctly clustered if two nodes that belong to one community in the ground truth are assigned to the same community in the result, and if two nodes from different communities are assigned to different communities. Let Boolean function $G(i, j)$ denote whether, in the ground truth, node i and node j are in the same community, and $C(i, j)$ denote whether, in the clustering result, node i and node j are in the same cluster. A pair is correctly clustered iff $G(i, j) = C(i, j)$. Note that this measure does not require a direct correspondence of clusters, but measures to what degree pairs of lines of changed code in a simulated fork are correctly assigned to the same or differing features.

Independent variables. For RQ1 and RQ2, we compare the accuracy of Infox when changing which subsets of edges to consider and whether to perform splitting and joining. As automated stop criteria for splitting, we stop after 5 additional clusters; for joining,

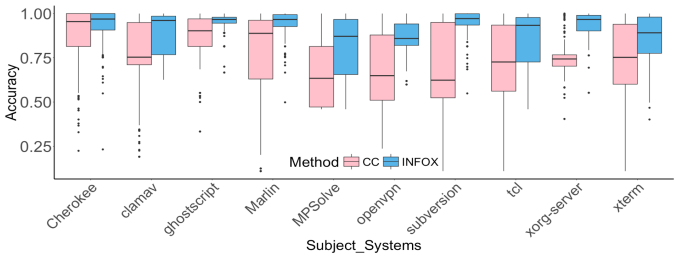


Figure 7: Accuracy of Infox and ClusterChanges (CC) for 10 projects, 156 simulated forks units each.

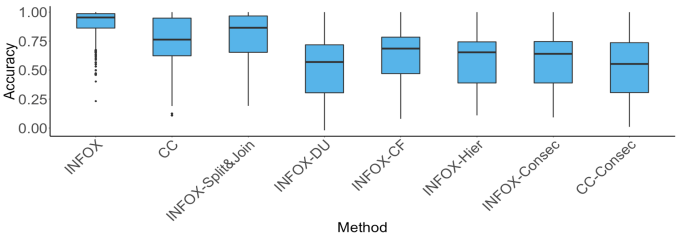


Figure 8: Accuracy across all 1560 simulated forks for different variations.

we use our default stop criterion described in Section 3.2. Furthermore, we consider how ClusterChanges would perform if used for this problem unmodified (conceptually equivalent to Infox without further splitting and joining and limited to consecutive lines and def-use dependencies in the clustering process). We used the Paired Wilcoxon rank-sum test to establish statistic significance.

Threats to validity. External validity is bound by the use of simulated forks, that provide ground truth for realistic settings but are not real forks. The elimination of nested macros may make simulated forks to be cleaner than real forks. Nonetheless, we select systems from different domains with different number of macros which are heavily based on industry-strength technologies. Besides, we did not rely on the ifdef evaluation alone, but triangulated our results with the user study (see Sec. 5.2). INFOX is conceptually entirely independent of the programming language. With respect to implementation, generalizations to other languages than C/C++ should be done with care.

Regarding internal validity, our reimplementation of ClusterChanges may not be faithful, but was unavoidable as the original tool is not publicly available. To keep the design space manageable we do not explore different weights and stop criteria, but have only done initial sensitivity analyses to establish that the results are robust with regards to other weights or minor changes in stop criteria.

Results. We show the accuracy results in Fig. 7 aggregated over 1560 simulated forks of all 10 subjects. **Regarding RQ1, we conclude that Infox assigned features with 90 % accuracy and improves accuracy over ClusterChanges by 54-92%.** The results are stable across all 10 projects and statistically significant ($p < .05$). In 102 simulated forks (6.5%), Infox achieves a much higher accuracy than ClusterChanges (e.g,. accuracy increased 0.5), of which 61 cases are due to splitting and 41 cases are due to joining.

In Fig. 8, we show accuracy results of all 1560 simulated forks split by different variations, specifically different kinds of dependency

edges and with and without splitting and joining. **Regarding RQ2, we observe that splitting & joining steps improves accuracy by 4-14 %** (stat. sign., $p < .05$). Removing any kind of edges from the clustering approach significantly affects accuracy as well ($p < .05$); **all kinds of edges are important for the clustering quality**, but the definition-usage edges are the most influential ones.

5.2 Human-subject study (RQ3 & RQ4)

To evaluate the usability of INFOX, we contacted open-source developers who maintain forks to validate identified features and explore whether the generated summaries provide meaningful insights.

Study design. We invited developers of active forks (see selection below) for a remote interview. We conducted each interview in a semi-structured fashion divided into four phases:

- *Opening and introduction:* We started each interview by briefly explaining our research topic and the general purpose of our study. We asked whether the participants would share their screen with us and whether they consent to screen and audio recording.
- *Validating clustering result (RQ3):* In order to help participants remember what the code changes are, and also help us to gain domain knowledge for a better conversation, we first asked participants to briefly describe the project and code changes. Subsequently, we sent them the clustering result of Infox for their own fork as a folder of HTML files (as illustrated in Fig. 5). Within those results, participants could split and join clusters interactively. We started with an initial clustering result (without any splitting) and explained how to read and navigate the results.
  In a subsequent discussion, we pursued two questions: Whether the keywords are representative of their feature implementation and whether the clustering of the source code is meaningful to them. Most of the participants were communicative, and right after spending some time learning how to interact with Infox, they started to navigate among code changes, explaining the meaning of the code, and whether clusters made sense or not. In line with methods for think-aloud protocols [42], we encouraged participants that were interacting with INFOX without saying anything for a long time to speak out loud, asking probing questions, such as *"Could you tell us what are you looking at?"* or *"Would you explain what this code cluster means?"*
- *Exploring the project overview (RQ4):* Before exploring Infox's summary of other forks, we transitioned the discussion with the question *"Do you check what other forks are doing in this project?"* and followed up with questions on how and for what purpose they do this. Afterward, we sent them the project overview (cf. Fig. 1b) and encouraged them to look through the list of forks. By clicking on the name of a fork, they could also explore that fork's code with Infox's results, just as they previously did for their own fork. Participants were usually actively exploring other forks at this point without our prompting and shared discoveries with us. When participants explored the code of a fork, we asked whether the keyword summary provided them a reasonable approximation of what they found in the implementation. In addition, we opportunistically asked questions about the relevance of keywords and the accuracy of clustering results in other forks based on their understanding (similar to questions about their own fork previously) when it fit the flow of the exploration.
- *Open discussion and closing:* We concluded each session with general and open-ended questions about further use cases and suggestions for improvement.

We compensated each participants with a $10 Amazon gift card. The interviews lasted between 30 and 90 minutes.

Participant selection. We searched for projects with active forks using two strategies. First, we used the GitHub search to find projects written in C/C++, selecting projects with more than 30 forks. Second, we queried GHTorrent [36] for the 100 C/C++ projects with the most first-level forks.

Among these projects, we selected forks that: (a) had at least one commit within the last year (increasing the chance that interviewees can remember their changes), (b) have added at least 10 lines of code (smaller changes are less likely to be a feature implementation), (c) have a large portion of commits submitted by the fork owner (excluding forks that aggregate changes of others), and (d) have a public email address or website of the fork owner. To enable questions about the overview page, we excluded projects for which we could not find at least three forks that fit these criteria.

In the end, we analyzed 58 projects on Github and found 12 projects fit our filtering criteria. We identified 81 fork owners. We sent out an email to candidate developers briefly describing our study. We interviewed 11 developers from 7 different projects (response rate 13.6%). We quickly reached saturation in that additional interviews provided only marginal additional insights. In Table 2 we list the characteristics of the projects from which we interviewed developers. All developers are experienced open-source developers.

Analysis. We analyzed the interviews primarily qualitatively, analyzing what participants learned and how they interacted with the tool. Two of the authors transcribed and coded the interviews, following standard methods of qualitative empirical research [66].

Threats to validity. Regarding external validity, our study may suffer from a selection bias, as common for these kinds of studies. Many of our participants work on 3D printers, which may have different characteristics. However, overall we reached developers from several different domains and did not observe any systematic differences. Finally, we focus on open source whereas results may differ in industrial settings in which forks are centrally managed.

Regarding internal validity, communication issues may have affected some answers; we mitigated this threat by refining our interview guide when questions raised confusion and involved two researchers in each interview. Despite open-ended questions and careful design (see above), we cannot entirely exclude confirmation bias, in which participants might avoid raising critical points; we mitigate this by focusing on insights gained, not just claims.

Results. Regarding RQ3 (clustering quality), participants mostly confirmed that the clustering results were appropriate, but often fine-tuned them with further splitting and joining. This further supports the need for interactive tools. Overall, participants supported our decision to cluster changes in a fork. For example, participant P4 said: *"It is necessary to split code changes into pieces, even though they cannot be executed in isolation."* Of the 11 participants, 10 said

Table 2: Participants of our user study and their projects

Project	#Forks	#Active Forks	LOC	Change size (LOC)	Domain	Participant
MarlinFirmware/Marlin	4149	1901	19,799	2-3753	3D printer	P1 P3
Smoothieware/Smoothieware	566	237	61,425	19-11, 263	3D printer	P5 P6 P7
grpc/grpc	2226	470	95,838	3-480,901	general-purpose RPC framework	P2
timscaffidi/ofxVideoRecorder	60	24	611	7-23,228	multi-threaded video recording extension	P4
arduino/Arduino	5592	669	112,692	23-7,643	electronic prototyping platform	P4
bitcoin/bitcoin	9696	1242	99,746	6-647	experimental digital currency	P8 P9 P10
ariya/phantomjs	4,921	749	10,031	45-2,358	Scriptable Headless WebKit	P11

that INFOX correctly identified the clusters most of the time, although there are small clusters (containing one or two lines) should have been merged into bigger clusters. The remaining participant pointed out a cluster containing unrelated code that was automatically generated by libraries and should be removed.

As we discussed earlier, INFOX provides flexibility to developers by allowing them to split or join clusters interactively. During the interviews, participants compared the splitting and joining results carefully, and after several steps, they usually identified clusters that they agreed with. For example a typical interaction flowed as follows, here from participant P5: *"I think this blue and yellow cluster should belong together.. [clicks the join button] ..oh, so your software correctly identifies all of this being one thing not two different things."*

The participants identified some cases in which the clustering result could be improved, usually caused by technical limitations of the dependency analysis in our prototype (see Section 7). For example, when P4 found a 1-line cluster that should belong to another bigger cluster, the participant said: *"I know it is related, acceleration and volumetric (are related), but looking at just the syntax it is not, it is not using the same words. Adding check-box to manually merge selected clusters (could solve this problem)".*

In summary, **participants generally agreed that INFOX could identify correct clusters at certain splitting or joining steps** (RQ3). Participants suggested that INFOX could provide more flexibility for manually refining the clustering result. Even though limited to few participants, our interviews corroborate the high-accuracy results from our quantitative study in a realistic setting.

With regard to RQ4 (overview), we looked particularly for signs that developers learned new insights while exploring the overview. Of the 11 participants, we showed 10 participants (P2-P11) the overview of forks in their project and 8 gained different kinds of new information from the overview page:

- *Finding redundant development.* Two participants found other forks that are working on the same feature implementation as they did before. When they found these instances of redundant development, they explored the fork's source code. For example, P3 said :*"It does look like somebody did a very simple one-function [...] system. I think they should use our code, there is great reason to use it."* After skimming the overview page, P4 said: *"I can see multiple forks are working on the similar problem. This one looks like it is adding [...] that I already added."*
- *Find interesting and potentially reusable feature.* When skimming all the forks, 6 participants identified specific features of interest; For example, P5 expressed *"this is all laser stuff, this is useful."* When participants mentioned something is interesting, we asked them why. The answers all identify features that are important to the project or that they could reuse in their own forks, such as P5's statement *"If it is only exists in this fork, then I want to somehow get this fork into my fork."*

Beyond these specific actionable insights, many participants more generally indicated that this overview would be useful: By looking at the overview page, our participants found many forks that they did not know before, and by reading the summary table of each fork, they usually got the idea of what has happened in each fork. For example, participant P3 said: *"It is going to make it a lot easier to find the things you are looking for as a programmer."* and P6 explained *"I see all the differences for all the forks. Basically it is the same thing I am doing through GitHub, (but) only it is summarized in the same place, I don't have to jump and open 50 tabs to do it."* Participant P7 expressed interest to use the tool for another project he maintained, for which he always wanted to know what is going on in forks, but was limited by current tools.

Regarding labels for code they did not know, we could observe that they clearly gave some initial idea to participants and could typically describe what they would expect from the implementation. For example, participant P5 described *"the [keywords] give me some clues of temperature; I know which part of Smoothie is modified."* Overall, all participants thought the interpretation of keywords is similar to their understanding of the source code.

In summary, even though we interviewed only a small number of participants, **we found frequent and concrete evidence of new insights gained from the overview page, including redundant development and reusable contributions** (RQ4). This is encouraging for the usefulness of the approach and its capability to provide actionable insights.

6 RELATED WORK

Transparency in social coding. Transparency in modern social coding platforms has been shown to be essential for decision making in fast paced and distributed software development [19, 20]. Visible clues, such as developer activities or project popularity, influence decision making and reputation in an ecosystem. With this work, we make often-lost contributions in forks and branches transparent to developers with the aim of reducing inefficiencies in the development process.

Forking Practices. Before the rise of social coding, forking traditionally referred to splitting off a new independent development branch, to compete with or supersede the original project. The right for such hard forks (codified in open source licenses) was seen as essential for guaranteeing freedom and useful for fostering disruptive innovations [31, 55, 56], but hard forks themselves were often seen as antisocial and as risk to projects [31, 45, 55, 60]. In the context of modern forking, a lower bar of forking may encourage developers to maintain multiple variants of a product in parallel,

often not intended as hard forks. Gousios et al. explored GitHub's pull-request model, in which forking is an essential component. Their work confirms that forking provides increased opportunities for community engagement, but also highlights that only few contributions are integrated and pull requests are frequently rejected due to redundant development and missing coordination [35].

Understanding branches and forks. Conceptually closest to our work is Bird and Zimmerman's analysis of branches at Microsoft, revealing that too many branches can be an issue and *what-if* analysis to explore the costs of merging can support decision making [7]. In addition, several studies have studied forking practices in open source and industrial product line development [23, 52, 64, 71]. Those studies have revealed the discussed problems, but did not provide any solutions.

Untangling code changes. Technically, our work relates to work on untangling code changes. Originally, untangling code changes was driven by biases in mining repositories and predicting defects [21, 37]. Barnett et al. [5] proposed ClusterChanges to decompose tangled code changes in order to identify independent parts of changes, especially large commits, to facilitate understanding during the code reviewing process. A key assumption is that commits are not always cohesive and reliable. These approaches often analyze dependencies within a change and our implementation was inspired by and improves upon ClusterChanges, as discussed and evaluated.

Other strategies have been explored to untangle changes, including *semantic history slicing* that compares test executions [47, 48], and *EpiceaUntangler* [21] and *Thresher* [73] which interact with developers when committing a change, to encourage more cohesive commits. All these approaches are less applicable in our setting, as they would require test cases for all added functionality or upfront clean commits by all developers. In fact, Herzig and Zeller [38] argue that tangled changes are natural and should not be forbidden; we support this view and build tooling that extracts features after the fact, but at much larger granularity of differences in forks.

Concern location. Concern location (or concept or feature location) is the challenge of identifying the parts of the source code that correspond to a specific functionality, typically for maintenance tasks [59]. Based on a keyword or entry-point, developers or tools attempt to identify all code relevant for that feature. Concern location typically uses either a static, a dynamic, or an information-retrieval strategy [22, 76]: Static analyses examine structural information such as control or data flow dependencies [11, 62], whereas dynamic analyses examine the system's execution [15, 26]. In contrast, information-retrieval-based analyses perform some sort of search based on keywords [13, 22, 33, 50, 57] with more or less sophisticated natural language processing [41, 68]. Combinations of these strategies are common [22]. Our analysis has similarities with static concern-location approaches, but the setting is different: Instead of identifying code related to a specific given code fragment in a single code base, we aim at dividing the difference between two snapshots into cohesive code fragments without starting points. Whereas location usually identifies one concern at a time, we identify multiple features in a fork. At the same time, if execution traces or external keywords were available, those could likely be integrated into a clustering process like Infox.

Code summarization. Finally, there are many approaches to summarize source code [44, 53, 58, 70] using information retrieval to derive topics from the vocabulary usage at the source code level. So far, we use only a standard lightweight information-retrieval technique to identify keywords for clusters, but combinations with more advanced summarization strategies might improve results significantly.

7 DISCUSSION AND CONCLUSION

Evidence from both academia and industry shows that current fork-based development is popular but has many practical problems that can be traced to a lack of transparency. Because developers do not have an overview of forks of a project, problems like redundant development, lost contributions and suboptimal forking point arise. To improve the transparency, we designed an approach to identify features from forks and generate an overview of the project in order to inform developers of what has happened in each active fork.

Infox is a first step in making transparent what happens in forks of a project, and it can be a building block in a larger endeavor to support fork-based development, such that it keeps its main benefits, such as ease of use and distributed and independent development, while addressing many of its shortcomings through tool support.

This new transparency, might address problems including lost contributions and redundant development. All participants in our human subject study had immediate ideas of who might benefit from such a tool, including *"the person who maintains the main branch"* [P4] and *"it is super useful for everybody, especially for major main Smoothieware developers"*[P6]. In addition our evaluation has shown that clustering results are accurate (90 % on average) and labels are meaningful summaries.

At the same time, Infox is just an initial prototype with technical limitations and many opportunities for extensions:

- The initial clustering strategy as well as the community-detection algorithm [34] are designed to divide a change into disjoint clusters. Boundaries between features are not always easy to define and features may overlap or may be split into subfeatures. Exploring other network analysis techniques to identify overlapped features or sub-features is an interesting avenue for further research.
- Although our clustering approach achieved high accuracy results, it would be worth to explore additional information that might provide insights about relationships of code fragments (even if unreliable generally), such as data-flow dependencies, syntactic or structural similarity between code fragments, code fragments that have been changed together in the same commit or by the same author. To identify which of these provide useful insights and which just create more noise.

ACKNOWLEDGMENTS

Kästner and Zhou's work has been supported in part by the National Science Foundation (awards 1318808, 1552944, and 1717022) and AFRL and DARPA (FA8750-16-2-0042). Stănciulescu's work has been supported in part by the EliteForsk travel scholarship from the Danish Ministry of Higher Education and Science. Xiong's work has been supported in part by National Key Research and Development Program 2016YFB1000105.

REFERENCES

[1] 2016. Dear Github Issue 109: Tell us Concisely What Other People Changed in Their Forks. (2016). https://github.com/dear-github/dear-github/issues/109

[2] 2016. Dear Github Issue 175: Better overview over forks. (2016). https://github.com/dear-github/dear-github/issues/175

[3] 2017. Lovely Forks Browser Extension: Show notable forks of Github repositories under their names. (2017). https://github.com/musically-ut/lovely-forks

[4] Michał Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Laemmel, Stefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. 2014. Flexible Product Line Engineering with a Virtual Platform. In *Comp. Int'l Conf. Software Engineering (ICSE)*. ACM, 532–535.

[5] Mike Barnett, Christian Bird, Joao Brunet, and Shuvendu K Lahiri. 2015. Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets. In *Proc. Int'l Conf. Software Engineering (ICSE)*, Vol. 1. IEEE, 134–144.

[6] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M Atlee, Krzysztof Czarnecki, and Andrzej Wąsowski. 2014. Three Cases of Feature-based Variability Modeling in Industry. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MoDELS)*. Springer, 302–319.

[7] Christian Bird and Thomas Zimmermann. 2012. Assessing the Value of Branches with What-if Analysis. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*. ACM, 45.

[8] Jürgen Bitzer and Philipp JH Schröder. 2006. The Impact of Entry and Competition by Open Source Software on Innovation Activity. *The economics of open source software development* (2006), 219–245.

[9] Jan Bosch. 2009. From Software Product Lines to Software Ecosystems. In *Proc. Int'l Software Product Line Conf. (SPLC)*. Carnegie Mellon University, 111–119.

[10] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2011. Improving the Tokenisation of Identifier Names. *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)* (2011), 130–154.

[11] Kunrong Chen and Václav Rajlich. 2000. Case Study of Feature Location using Dependence Graph. In *Proc. Int'l Workshop on Program Comprehension (IWPC)*. IEEE, 241–247.

[12] Bredan Cleary and Chris Exton. 2007. *Assisting Concept Location in Software Comprehension.* Ph.D. Dissertation. University of Limerick.

[13] Brendan Cleary, Chris Exton, Jim Buckley, and Michael English. 2009. An Empirical Analysis of Information Retrieval based Concept Location Techniques in Software Comprehension. *Empirical Software Engineering* 14, 1 (2009), 93–130.

[14] Michael L Collard, Michael John Decker, and Jonathan I Maletic. 2013. srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration. In *Proc. Int'l Conf. Software Maintenance (ICSM)*. IEEE, 516–519.

[15] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. 2009. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Trans. Softw. Eng. (TSE)* 35, 5 (2009), 684–702.

[16] Davor Čubranić and Gail C Murphy. 2003. Hipikat: Recommending Pertinent Software Development Artifacts. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Computer Society, 408–418.

[17] Davor ČubraniĆ, Gail C Murphy, Janice Singer, and Kellogg S Booth. 2004. Learning from Project History: a Case Study for Software Development. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)*. ACM, 82–91.

[18] Davor Cubranic, Gail C Murphy, Janice Singer, and Kellogg S Booth. 2005. Hipikat: A Project Memory for Software Development. *IEEE Trans. Softw. Eng. (TSE)* 31, 6 (2005), 446–465.

[19] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)*. ACM, 1277–1286.

[20] Laura Dabbish, Colleen Stuart, Jason Tsay, and James Herbsleb. 2013. Leveraging Transparency. *IEEE Software* 30, 1 (2013), 37–43.

[21] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. 2015. Untangling fine-grained code changes. In *Proc. Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 341–350.

[22] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature Location in Source Code: a Taxonomy and Survey. *Journal of software: Evolution and Process* 25, 1 (2013), 53–95.

[23] Yael Dubinsky, Julia Rubin, Theodore Berger, Slawomir Duszynski, Matthias Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Pines. In *Proc. Europ. Conf. Software Maintenance and Reengineering (CSMR)*. IEEE, 25–34.

[24] Anh Nguyen Duc, Audris Mockus, Randy Hackbarth, and John Palframan. 2014. Forking and Coordination in Multi-platform Development: A Case Study. In *Proc. Int'l Symp. Empirical Software Engineering and Measurement (ESEM)*. ACM, 59:1–59:10.

[25] Marc Eaddy, Alfred V Aho, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2008. Cerberus: Tracing Requirements to Source Code using Information Retrieval, Dynamic Analysis, and Program Analysis. In *Proc. Int'l Conf. Program Comprehension (ICPC)*. Ieee, 53–62.

[26] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. 2003. Locating Features in Source Code. *IEEE Trans. Softw. Eng. (TSE)* 29, 3 (2003), 210–224.

[27] Thomas J Emerson. 1984. A Discriminant Metric for Module Cohesion. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Press, 294–303.

[28] Michael D. Ernst, Greg J. Badros, and David Notkin. 2002. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. Softw. Eng. (TSE)* (2002), 1146–1170.

[29] Neil A Ernst, Steve Easterbrook, and John Mylopoulos. 2010. Code Forking in Open-source Software: a Requirements Perspective. *arXiv preprint arXiv:1004.2889* (2010).

[30] Janet Feigenspan, Maria Papendieck, Christian Kästner, Mathias Frisch, and Raimund Dachselt. 2011. FeatureCommander: Colorful #ifdef World. In *Proc. Int'l Software Product Line Conf. (SPLC)*. ACM, 48.

[31] Karl Fogel. 2005. *Producing Open Source Software: How to Run a Successful Free Software Project.* " O'Reilly Media, Inc.".

[32] Santo Fortunato. 2010. Community Detection in Graphs. *Physics reports* 486, 3 (2010), 75–174.

[33] Gregory Gay, Sonia Haiduc, Andrian Marcus, and Tim Menzies. 2009. On the Use of Relevance Feedback in IR-based Concept Location. In *Proc. Int'l Conf. Software Maintenance (ICSM)*. IEEE, 351–360.

[34] Michelle Girvan and Mark EJ Newman. 2002. Community Structure in Social and Biological Networks. *Proceedings of the national academy of sciences* 99, 12 (2002), 7821–7826.

[35] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An Exploratory Study of the Pull-based Software Development Model. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 345–355.

[36] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. 2014. Lean GHTorrent: GitHub Data on Demand. In *Proc. Int'l Conf. Mining Software Repositories (MSR)*. ACM, 384–387.

[37] Kim Herzig and Andreas Zeller. 2011. Untangling Changes. *Unpublished manuscript, September* 37 (2011), 38–40.

[38] Kim Herzig and Andreas Zeller. 2013. The Impact of Tangled Code Changes. In *Proc. Int'l Conf. Mining Software Repositories (MSR)*. IEEE Press, 121–130.

[39] Kim Herzig and Andreas Zeller. 2013. The Impact of Tangled Code Changes. In *Proc. Int'l Conf. Mining Software Repositories (MSR)*. IEEE, 121–130.

[40] Emily Hill, Lori Pollock, and K Vijay-Shanker. 2007. Exploring the Neighborhood with Dora to Expedite Software Maintenance. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. ACM, 14–23.

[41] Emily Hill, Lori Pollock, and K Vijay-Shanker. 2009. Automatically Capturing Source Code Context of NL-queries for Software Maintenance and Reuse. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE, 232–242.

[42] Riitta Jääskeläinen. 2010. Think-aloud Protocol. *Handbook of translation studies* 1 (2010), 371–374.

[43] David Kawrykow and Martin P. Robillard. 2011. Non-essential Changes in Version Histories. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 351–360.

[44] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba. 2007. Semantic Clustering: Identifying Topics in Source Code. *Information and Software Technology (IST)* 49, 3 (2007), 230–243.

[45] Andrew M St Laurent. 2004. *Understanding Open Source and Free Software Licensing: Guide to Navigating Licensing Issues in Existing & New Software.* " O'Reilly Media, Inc.".

[46] Sungjick Lee and Han-joon Kim. 2008. News Keyword Extraction for Topic Tracking. In *Proc. Int'l Conf. Networked Computing and Advanced Information (NCM)*. IEEE, 554–559.

[47] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. 2016. Precise Semantic History Slicing through Dynamic Delta Refinement. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. 495–506.

[48] Y. Li, C. Zhu, J. Rubin, and M. Chechik. 2017. Semantic Slicing of Software Version Histories. *IEEE Trans. Softw. Eng. (TSE)* (2017), 1–1.

[49] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-based Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*.

[50] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I Maletic. 2004. An information retrieval approach to concept location in source code. In *Proc. Working Conf. Reverse Engineering (WCRE)*. IEEE, 214–223.

[51] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. 2015. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 495–518.

[52] Tommi Mikkonen and Linus Nyman. 2011. To Fork or Not to Fork: Fork Motivations in SourceForge Projects. *Int. J. Open Source Softw. Process.* 3, 3 (July 2011), 1–9.

[53] Gail Cecile Murphy. 1996. *Lightweight Structural Summarization as an Aid to Software Evolution.* Ph.D. Dissertation.

[54] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How We Refactor, and How We Know It. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Computer Society, 287–297.

[55] Linus Nyman. 2014. Hackers on forking. In *Proceedings of The International Symposium on Open Collaboration*. ACM, 6.

[56] Linus Nyman, Tommi Mikkonen, Juho Lindman, and Martin Fougère. 1999. Perspectives on Code Forking and Sustainability in Open Source Software. *Why Linux on't fork* (1999). http://linuxmafia.com/faq/Licensing_and_Law/forking.html.

[57] Maksym Petrenko, Václav Rajlich, and Radu Vanciu. 2008. Partial Domain Comprehension in Software Evolution and Maintenance. In *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE, 13–22.

[58] Denys Poshyvanyk and Andrian Marcus. 2007. Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. In *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE, 37–48.

[59] V. Rajlich and N. Wilde. 2002. The Role of Concepts in Program Comprehension. In *Proc. Int'l Conf. Program Comprehension (ICPC)*. 271–278.

[60] Eric S Raymond. 2001. *The Cathedral & the Bazaar: Musings on linux and open source by an accidental revolutionary.* " O'Reilly Media, Inc.".

[61] Meghan Revelle, Bogdan Dit, and Denys Poshyvanyk. 2010. Using Data Fusion and Web Mining to Support Feature Location in Software. In *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE, 14–23.

[62] Martin P Robillard. 2005. Automatic Generation of Suggestions for Program Investigation. In *SIGSOFT Softw. Eng. Notes*, Vol. 30. ACM, 11–20.

[63] Martin P Robillard, David Shepherd, Emily Hill, K Vijay-Shanker, and Lori Pollock. 2007. An Empirical Study of the Concept Assignment Problem. *School of Computer Science, McGill University, Tech. Rep. SOCS-TR-2007.3* (2007).

[64] Gregorio Robles and Jesús M. González-Barahona. 2012. A Comprehensive Study of Software Forks: Dates, Reasons and Outcomes. In *Open Source Systems: Long-Term Sustainability - 8th IFIP WG 2.13 International Conference, OSS 2012, Hammamet, Tunisia, September 10-13, 2012. Proceedings*. 1–14.

[65] Julia Rubin and Marsha Chechik. 2013. A Framework for Managing Cloned Product Variants. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Press, 1233–1236.

[66] Johnny Saldaña. 2015. *The Coding Manual for Qualitative Researchers*. Sage.

[67] Gerard Salton and Christopher Buckley. 1988. Term-weighting Approaches in Automatic Text Retrieval. *Information processing & management* 24, 5 (1988), 513–523.

[68] David Shepherd, Zachary P Fry, Emily Hill, Lori Pollock, and K Vijay-Shanker. 2007. Using Natural Language Program Analysis to Locate and Understand Action-oriented Concerns. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*. ACM, 212–224.

[69] Igor STEINMACHER, Gustavo H. L. PINTO, Igor Scaliante WIESE, and Marco Aurélio GEROSA. 2018. Almost There: A Study on Quasi-Contributors in Open-Source Software Projects. In *Proc. Int'l Conf. Software Engineering (ICSE)*. 1–12.

[70] Margaret-Anne Storey, Li-Te Cheng, Ian Bull, and Peter Rigby. 2006. Shared Waypoints and Social Tagging to Support Collaboration in Software Development. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)*. ACM, 195–198.

[71] Ştefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. In *Proc. Int'l Conf. Software Maintenance (ICSM)*. IEEE, 151–160.

[72] Ching Y Suen. 1979. N-gram Statistics for Natural Language Understanding and Text Processing. *IEEE transactions on pattern analysis and machine intelligence* 2 (1979), 164–172.

[73] Marcel Taeumel, Stephanie Platz, Bastian Steinert, Robert Hirschfeld, and Hidehiko Masuhara. 2017. Unravel Programming Sessions with THRESHER: Identifying Coherent and Complete Sets of Fine-granular Source Code Changes. *Information and Media Technologies* 12 (2017), 24–39.

[74] Lei Tang and Huan Liu. 2010. Community Detection and Mining in Social Media. *Synthesis Lectures on Data Mining and Knowledge Discovery* (2010), 1–137.

[75] Greg R Vetter. 2007. Open Source Licensing and Scattering Opportunism in Software Standards. *BCL Rev.* 48 (2007), 225.

[76] Norman Wilde and Michael C Scully. 1995. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software: Evolution and Process* 7, 1 (1995), 49–62.

[77] Andrew Y Yao. 2001. CVSSearch: Searching through Source Code using CVS Comments. In *Proc. Int'l Conf. Software Maintenance (ICSM)*. IEEE Computer Society, 364.