How Can I Contribute? A Qualitative Analysis of Community Websites of 25 Unix-Like Distributions

Jacob Krüger
Otto-von-Guericke University
Magdeburg, Germany

Sebastian Nielebock
Otto-von-Guericke University
Magdeburg, Germany

Robert Heumüller
Otto-von-Guericke University
Magdeburg, Germany

ABSTRACT

Developers collaboratively implement large-scale industrial and open-source projects. Such projects pose several challenges for developers, as they require considerable knowledge about the project and its development processes, for instance, to fix bugs or implement new features. Understanding what information developer communities codify on how to contribute to their project is crucial, for example, to onboard new developers or for researchers to scope analysis techniques. In this paper, we report the results of a qualitative analysis of 25 Unix-like distributions, focusing on what information the communities codify publicly on contributing. The results reveal no dedicated strategies to codify information on contribution or development practices. Still, non-technical contributions are easy to identify, while information on the development is hard to collect—and mostly concerned with versioning and bug reporting. Our insights help to understand information-provisioning strategies, identify information sources, and scope analyses.

CCS CONCEPTS

• **Software and its engineering** → *Documentation.*

KEYWORDS

Linux, Unix, Information provisioning, Development, Bug fixing

ACM Reference Format:
Jacob Krüger, Sebastian Nielebock, and Robert Heumüller. 2020. How Can I Contribute? A Qualitative Analysis of Community Websites of 25 Unix-Like Distributions. In *Evaluation and Assessment in Software Engineering (EASE 2020), April 15–17, 2020, Trondheim, Norway.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3383219.3383256

1 INTRODUCTION

Most software is developed by a community of developers, is constantly increasing in size, has a high level of complexity, and its arguably impossible for a developer to know all of its parts [7, 13, 14, 17, 18]. Before working on such software, developers have to understand how they can contribute to the project: What are the rules and development practices? This knowledge is important to onboard developers, evolve the project in a consistent way, understand practices or scope analysis techniques [15, 28].

EASE 2020, April 15–17, 2020, Trondheim, Norway

ACM ISBN 978-1-4503-7731-7/20/04…$15.00
https://doi.org/10.1145/3383219.3383256

We are not aware of an empirical study analyzing what information communities codify and provide *publicly*, supporting especially externals to understand their practices. For instance, a community may use a Wiki, website or software-hosting platform (e.g., GitHub) for this purpose. So, the question is whether communities employ dedicated strategies, such as using a single place (e.g., a Wiki page) to provide codified information? This is an important question, as highlighted by Steinmacher et al. [28, 29], who identified a "Poor How to Contribute" overview and related issues (e.g., outdated documentation) as barriers for new developers. For researchers, this information is important, for example, to verify findings, adopt techniques, and select subject systems.

In this paper, we report a qualitative analysis of 25 Unix-like distributions. These distributions, are complex (e.g., the Linux Kernel and packages), have large communities, are often long-living, and are open-source—usually encouraging developers to contribute. For instance, the following statement sketches the complexity of one distribution, indicating various roles, practices, and packages:

> Arch
> "Arch developers and Trusted Users are responsible for compiling, packaging, and distributing software from a wide range of sources."
> https://wiki.archlinux.org/index.php/Bug_reporting_guidelines

So, Unix-like distributions allow us to investigate the practices of different developer communities that should aim to codify and provide information publicly. We conducted an extensive manual study of the communities' infrastructures based on a mix of qualitative document analysis [5], open-card sorting [33], and open coding [9]. In detail, our contributions are:

- We identify how 25 communities codify and provide information on how to contribute to their projects for externals.
- We describe and classify different types of contributions.
- We analyze feature-development and bug-reporting practices.
- We provide an open-access replication package.[1]

The results indicate that Unix-like distributions are highly diverse in their information provisioning. This means that new developers—and researchers—face severe challenges to familiarize with a distribution and its community, particularly since many details seem to be not codified or only available to selected developers.

2 RELATED WORK

Knowledge Sharing. Rodríguez et al. [25] found that collaborative work enforces developers to share knowledge, and that the availability of information must also be shared or it will be lost. Hemetsberger and Reinhardt [11] investigated how OSS-communities enable the re-experience of knowledge, which is based on code documentation and the repository. Analyzing the R community, Vasilescu

[1] https://doi.org/10.5281/zenodo.3665429

et al. [32] identified that developers shift towards community question answering systems [27], as they receive faster feedback compared to mailing lists. In contrast to these works, we strove for a broader view regarding how information is codified and provided.

OSS/FLOSS Communities. The main motivation for frequently contributing to an OSS/FLOSS project is increased social capital [12, 23], such as identifying with the project, solidarity with other developers, personal career, learning new technologies or enjoyment. Roberts et al. [24] analyzed the relation among such motivations, for example, getting paid as an extrinsic incentive may negatively impact the intrinsic motivation to identify with the project. Factors for disengagement in OSS/FLOSS projects are gender biases [10, 20, 31], not accepting pull requests from externals [22], missing guidelines [6], problems setting up the development environment as well as communication issues [20], insufficient mentoring [3], and personal changes [21]. We contribute a complementary analysis of such problems (e.g., missing guidelines) focusing on a problem mentioned, but not analyzed, by Steinmacher et al. [28, 29]: a "Poor How To Contribute" section.

OSS/FLOSS Projects. Researchers use OSS/FLOSS projects as subject systems, for instance, to analyze differences in success criteria [8], quality management [2], and development practices [15, 26] compared to other projects. Particularly, the Linux Kernel is used regularly to examine the characteristics of software systems. For example, this includes variability analysis and bugs [1, 14], code evolution [13, 19], and software aging [7]. In contrast to such works, we focus on the information provisioning, with our findings supporting researchers to scope and improve their analyses.

3 METHODOLOGY

Research Questions. Our goal was to capture what information on contributing communities codify and provide publicly. To this end, we defined four research questions:

RQ$_1$ *How do communities provide information?*
RQ$_2$ *How can developers contribute to distributions?*
RQ$_3$ *What feature-developmenmt practices are publicly codified?*
RQ$_4$ *What bug-reporting practices are publicly codified?*

Our analysis focuses on information that the communities make publicly available. So, we did not register to websites, contact community members or try to gain developer status, meaning that there may be additional information available. This represents the perspective of developers, users, and researchers who consider joining or using a project and aim to get informed beforehand.

Subject Systems. We conducted our analysis on 25 Unix-like distributions. Mostly, these are Linux distributions that build on the Linux Kernel—on its own one of the largest and most complex systems [13, 14, 30]—and extend it with own and foreign packages to deliver an operating system. An established distribution is usually developed by a larger community and the system's complexity challenges development (e.g., resolving conflicts between packages).

To avoid subjectivity and sampling biases, we selected our sample from DistroWach, a website that collects data on Unix-like distributions. We considered the highest ranked distributions as of 2018,[2] the most recently completed year at the time of our analysis. The "Hits Per Day" ranking that DistroWatch uses is not ideal, but reflects the interest of the site's visitors in different distributions based on how often the corresponding entries were visited. HPD is rather objective compared to an authors' sample and more interesting for developers than a random sample. As we show in Table 1, we included major (e.g., Ubuntu) and specialized (e.g., Kali) distributions. We also summarize the ranking of each distribution and the results of our analysis, excluding one distribution that was not listed as active anymore (i.e., #11: Antergos).

[2]https://www.distrowatch.com/index.php?dataspan=2018

Table 1: Overview of the 25 distributions we analyzed.

#	Name	HPD	UA	GI	D			QA		
					DP	FD	FR	BR	RP	TP
1	Manjaro	3778	●		●	●	●	●		
2	Mint	2495	●	●	●	●	●	●		●
3	elementary	1708	●	●	●	●	●	●		
4	MX Linux	1694		●		●		●		
5	Ubuntu	1506	●			●	●	●		●
6	Debian	1259	●		●	●	●	●	●	●
7	Solus	916	●	●	●	●	●	●		
8	Fedora	900	●	●	●	●	●	●	●	●
9	OpenSUSE	768	●	●	●	●	●	●		●
10	ZorinOS	642			●					
12	CentOS	596	●	●	●	●		●	●	●
13	Arch	580		●	●	●	●	●		●
14	ReactOS	547			●	●		●		●
15	Kali	514	●	●		●	●	●	●	
16	antiX	501						●		
17	KDE neon	499	●	●	●	●	●	●	●	●
18	TrueOS	497				●	●	●	●	
19	Lite	488		●		●		●		
20	Lubuntu	412			●	●		●	●	●
21	deepin	401	●	●		●	●	●		
22	PCLinuxOS	394								
23	Peppermint	368		●				●		●
24	Endless	365				●		●		●
25	FreeBSD	357	●	●	●	●	●	●	●	●
26	SmartOS	352		●	●	●		●	●	●
		Σ	**13**	**15**	**15**	**21**	**14**	**23**	**9**	**14**

UA: Unrestricted Accessibility; GI: Gathered Information
D: Development; QA: Quality Assurance; DP: Deployment Practices
FD: Feature Development and Versioning; FR: Feature Requests
BR: Bug Reporting; RP: Review Practices; TP: Testing Practices

Identifying Information. The community of each distribution maintains a website, providing information codified in natural language. Analyzing natural language is a laborious task that requires manual analysis. For this reason, we conducted a qualitative study following the recommendations of Bowen [5]. Initially, the first author manually inspected each DistroWatch entry and the websites linked there to identify inactive distributions and relevant content based on keywords, such as "Get Involved", "Join" or "Contribute". We then documented whether the information was gathered at a single place, whether it was accessible (i.e., **RQ$_1$**), and how developers could contribute (i.e., **RQ$_2$**). To classify different contribution options, the second author employed an iterative open-card sorting [33] based on textual summaries.

Afterwards, we read the descriptions of the two most common technical contribution practices—feature development and versioning (i.e., **RQ$_3$**) as well as bug reporting (i.e., **RQ$_4$**)—in more detail to identify what information on what level of detail the communities codified publicly. First, we analyzed what information each distribution provided regarding these practices, and also checked our previous results. As we dealt with unstructured and scattered information, we split the task among all authors, each analyzing

Table 2: Development and quality-assurance practices.

	Practice	Information on ...
D	Deployment (DP)	how to create binaries and distribute a new version (e.g., continuous integration).
D	Feature Development and Versioning (FD)	how to write code and use the version control system (e.g., branching strategies).
D	Feature Requests (FR)	how to propose new features and how features are selected (e.g., new packages).
QA	Bug Reporting (BR)	how to report bugs (e.g., in an issue tracker).
QA	Review (RP)	how code is reviewed (e.g., by core developers).
QA	Testing (TP)	how the distribution is tested (e.g., manual).

eight to nine distributions. We iterated through the links provided on each distribution's website to identify relevant information. This process was cross-checked by one other author for each distribution to check the results. Second, to answer **RQ$_3$** and **RQ$_4$**, the first two authors employed open coding on the resulting data. Afterwards, we used open-card sorting to define themes based on the identified codes. We documented each website that comprised valuable information and provide copies in our replication package.[1]

4 RQ$_1$: INFORMATION PROVISIONING

Results. For accessibility, we found that several communities employ access restrictions on their information. In some cases, developers have to fill in contact forms (e.g., MX Linux) or apply (e.g., Arch) to become a member. These access restrictions vary between communities and contribution types, for example, in most cases any user may report bugs, while few communities are completely open for developers to suggest any change, and some ask developers to contribute to another (base) distribution (e.g., antiX, Endless). These variations make a quantitative assessment problematic, but we found access restrictions of apparently existing information for nine distributions, while we seemed to have unrestricted access to all pieces of information for 13 distributions (i.e., UA in Table 1).

For gathering information, we found 15 communities that collect links at a dedicated place (CI in Table 1), referring to tools and guides, such as Bugzilla. Mostly, these centralized places are a website or Wiki page that is highlighted on the main page with keywords, such as "Get Involved". Seven distributions have only scattered information (e.g., information is on Wiki pages and discussion boards without links) and three apparently do not gather information publicly (i.e., ZorinOS, antiX, and Endless).

Discussion. Most communities employ their own information-provisioning strategy, ranging from openly publishing to completely hiding information. Clearly, some strategies can hamper the onboarding of new developers and the understanding of practices employed. While the information may be available after becoming part of a core development team, this is a high threshold and takes a lot of effort. Similarly, many communities do not have a dedicated place that explains how to contribute to a distribution.

Based on what we identified during our study, we hypothesize that most communities have good reasons for employing their practices. First, distributions emerge over time and their communities focus on implementing the distribution itself. So, they spend less time on codifying information. This may only happen if a distribution gains more traction or needs new developers, for example:

> TrueOS
>
> "The TrueOS Project has existed for over ten years. Until now, there was no formally defined process for interested individuals [...] to earn contributor status [...]. The current core TrueOS developers [...] wish to formalize the process [...]"
>
> https://www.trueos.org/contribute/

Second, communities with specialized distributions (e.g., Kali focuses on forensics) apparently aim to keep control over their distribution, having a dedicated team that decides about new features. This is highlighted by some communities' practices:

> TrueOS
>
> "[...] the core developers review the commit logs, removing elements that break the Project or deviate too far from its intended purpose."
>
> https://www.trueos.org/contribute/

Third, larger communities seem to aim to ensure quality by enforcing processes and restricting developers' access. For example, Debian and Arch have restrictive onboarding processes (e.g., assessing contributions to community projects). Finally, few distributions (e.g., ZorinOS) are developed by companies, selling some variants and restricting the access to detailed information.

RQ$_1$: How do communities provide information?

We did not identify dedicated strategies to codify and provide information publicly, but varying degrees of accessibility and gathering of information.

5 RQ$_2$: HOW TO CONTRIBUTE

Results. Most communities combine various tools (e.g., discussion boards, community-question-answering systems [4, 16, 27], and social media) and exemplify various options to contribute (e.g., reporting bugs, maintaining Wikis, and donating). During our open-card sorting, we identified three options for how and what developers can contribute. Due to the aforementioned limitations (**RQ$_1$**), we cannot quantify precisely how many communities employ what option. We again found variations regarding the information on what tools are used (e.g., GitLab, GitHub, and own versioning systems, such as OSC of OpenSUSE) for what option.

Non-Technical Contributions. We found several options to contribute without getting involved in development, mostly donating money (also called sponsoring/funding). Quite regularly, we found documenting, translating, and promoting a distribution; contributing to discussion boards, community-question-answering systems, and Wikis; as well as providing designs (e.g., websites, user interfaces) and artwork (e.g., Mint, Fedora, CentOS). Some distributions have rare options, for instance, supporting event organization (i.e., Fedora), a blog (e.g., Kali, Lubuntu) or a magazine (i.e., PCLinuxOS).

Technical Contributions. For technical contributions, we found considerably less information compared to non-technical ones. We identified five types of practices (cf. Table 2) during our first open-card sorting, and added *reviewing* (RP) while cross-checking the results and additional data. Interestingly, we observed that several distributions have separated repositories for external developers to implement own packages. Probably best known may be the Arch User Repository (AUR)[3] that allows to upload own packages that

[3] https://aur.archlinux.org/

users can compile and run on Arch and its derivatives (e.g., Manjaro). Moreover, the Arch community uses AUR contributions to judge which developers may join their development team. Similar practices exist for other distributions (e.g., Mint community projects) and some even allow their community to vote what packages should be included into the main distribution (e.g., Fedora).

Communication & Tools. Developers must communicate and use specific tools to contribute. Depending on the contribution, different channels are used by each community, including instant messengers, discussion boards, mailing lists, wikis, and social media. In fact, most distributions utilize multiple channels for communication and information-provisioning, making it harder to identify the ones that comprise valuable information (cf. **RQ$_1$**). For example, FreeBSD uses Bugzilla to report and fix bugs. However, the codified recommendation for contributing bug fixes is to read the corresponding mailing list—scattering information at two places and hiding information within communication.

Discussion. As for information-provisioning (cf. Section 4), we found numerous possibilities to contribute to a distribution. While non-technical contributions are often intuitively understandable and linked on the websites, information and tools for technical contributions are hard to identify. This problem challenges technical contributions, as it becomes unclear where to find information, identify open issues, and communicate or submit a contribution. Moreover, providing information and tooling at different places may easily result in inconsistencies and tangles communication with information. For example, some communities use mailing lists and Bugzilla in parallel, and Debian enforces templates for reporting bugs in mails that can also be send with the `reportbug` tool.[4] Such inconsistency may have simply evolved over time, is working well or represents a bad smell, but a more detailed understanding how different tools are used and combined in a community can help to improve our understanding of current practices.

Due to missing codified information and numerous options to contribute, it seems challenging for new users to onboard. Steinmacher et al. [28, 29] name specifically this problem as one barrier for new developers. However, we found an elegant solution by the Fedora community: An additional website[5] allows new users to interactively click through contribution options and links to corresponding Wiki pages. While this may not be the best solution, it is a unique one to motivate contributors, and comparing such an onboarding site with the usual plain websites seems interesting.

RQ$_2$: How can developers contribute to distributions?

We found various codified options for contributing to distributions, which are not limited to technical contributions. As different tools are used, information and communication may get scattered and tangled at places new developers do not know.

6 RQ$_3$: DEVELOPMENT PRACTICES

Results. As we show in Table 1, we found six codified development and quality-assurance practices. For deployment practices (DP) we only considered cases where we found information on how to build binaries from the source files. For feature requests (FR), we found that most distributions ask developers to suggest them either through an issue tracker (e.g., GitHub, Bugzilla) or pull requests (in GitHub). Few communities provide codified guidelines on how to submit and describe new features (e.g., Mint, Fedora). Interestingly, Fedora maintains a guideline on "forbidden" features, which excludes proprietary software, but also violations with US laws. The information codified most commonly (21) was on feature development and versioning, which we detail in the following.

Tools. The most common information we found was the *versioning* system used. In most cases, we found Git repositories or software-hosting platforms (e.g., GitHub, GitLab), while some communities rely (in addition) on SVN (e.g., FreeBSD), Launchpad and Bazaar (e.g., Ubuntu) or own systems (e.g., OpenSUSE). Interestingly, the Debian community provides extensive guides on how to develop, but supports only mirrors to download the source code, apparently not relying on a version control system. Most communities use the features of software-hosting platforms, particularly pull requests, to manage and trace contributions. Some communities specify additional *development tools*, such as distribution-specific libraries (e.g., GNOME for Mint), IDE recommendations (e.g., Sublime for Mint), review tools (e.g., Gerrit for deepin), build systems (e.g., Meson for elementary), and design tools (e.g., Glade for Mint). In some cases, we found scripts to check for style conformance (e.g., Solus).

Contributing. For contributing, several communities specify requirements and restrictions, also based on developer roles and the contributions an applicant has done. Some communities provide information on how developers can apply and what requirements they have to fulfill to *gain access*. To this end, communities have different access levels based on a developer's role. For example, FreeBSD defines the roles of "contributor", "committer", and "maintainer", each with specific responsibilities and access restrictions. In addition, most communities ask contributors to register to their systems, apply for access (e.g., via mail), get recommendations by members, maintain personal information, and participate regularly. For an application, the communities seem to focus on the applicant's *knowledge*. An applicant usually has to read, and agree to documents (e.g., guidelines, code of conduct), show that they are skilled in developing and the tools used, can cooperate with other developers (i.e., soft skills), and provide reference contributions (e.g., in the AUR). Concerning *development rules*, some communities define coding conventions. Some unique rules are that only individuals may receive access (i.e., FreeBSD) or that developers should not have access to the source code of specific software:

ReactOS

"We only ask that you have not had access to Microsoft source code for the area you want to work on."

https://www.reactos.org/participation

Finally, some communities codify how to *prepare* for a new contribution. Mainly, this includes how to check that the same contribution is not already under development and how to set up tools.

Soft Skills. We identified information on developers' soft skills for eight distributions. As these are complex, non-uniform and not about contributing features itself, we did not dig into detail. Still, it is an interesting finding that few communities seem to codify the required skill set of developers or the code of conduct employed. For

[4] https://www.debian.org/Bugs/Reporting
[5] https://whatcanidoforfedora.org/

example, Mint specifies understanding English as the only soft-skill needed, while Fedora defines a short code of conduct.[6]

Discussion. For most distributions, we were able to identify the (version control) system in which the community develops. While we also found recommendations on code styles and tools, the actual development process was rarely codified (cf. previous TrueOS examples). As a positive example, the Fedora[7] community describes the processes for various activities in detail. For some distributions, any developer can suggest features in issue trackers or pull requests, for example, the deepin community[8] reviews pull requests and acknowledges contributors after acceptance. However, it is unclear whether the same processes are employed by each community, as they may be hidden behind an application barrier. Further analyses seem necessary to assess what development practices should be codified to support the onboarding of new developers.

The application barrier seems quite high for several distributions, requiring significant knowledge about a distribution. Particularly challenging may be the regular contribution to packages, as for instance Arch (i.e., AUR) and TrueOS (i.e., five or more pull requests in six months) demand. Some communities still restrict a developers' activities according to specific roles. In contrast to these codified rules, soft skills are rarely mentioned. Overall, onboarding new developers seems rather problematic, as a lot of information is not codified. These problems arguably relate to our hypotheses for **RQ$_2$**, asking for a more detailed analysis of a community's motivations.

RQ$_3$: What feature-developmenmt practices are publicly codified?
Most distributions rely on a version control system, but we rarely found codified information on actual development practices and how to interact with the system.

7 RQ$_4$: QUALITY-ASSURANCE PRACTICES

Results. We found that nine communities codified information on reviewing (RP) and 14 on testing practices (TP). Some communities explicitly list becoming a tester as a way to contribute to the distribution. However, the processes are unclear, with few communities reporting about automated testing (e.g., Lubuntu, ReactOS) and testing teams (e.g., Arch). For reviewing, most communities only state that pull requests are reviewed, but not how this is done. We can see in Table 1 that almost all (23) communities describe how to report bugs (BR), so we focus on this practice in this section.

Tools. Many distributions use *issue trackers* to report bugs (and request features), particularly those integrated in software-hosting platforms. Still, we found third-party trackers, such as Launchpad, Bugzilla or Jira, and self-developed tools (e.g., Solus Development Portal[9]). If bugs are connected to third-party packages or base distributions, most communities ask to redirect the reports to those. Issue trackers usually incorporate *bug-report search engines*, which are explicitly mentioned and are used to avoid duplicate bug reports. Also, some communities use tools to *finance bug reports* to provide incentives for bug fixing, for example on Bountysource (e.g., Lite). Some communities use tools for *automated data collection and report creation*, for instance, `apport` (i.e., Ubuntu, Lubuntu) or `reportbug` (i.e., Debian). *Other tools* are sometimes mentioned to be used, but are actually not designed for this purpose. For instance, this includes mailing lists that involve automated processing of reports and forum threads for discussions. Again, tools get tangled, for example, Debian uses a mailing list based on which tickets are generated, while other distributions seem to just run both tools in parallel. Most tools are used to *assign and clarify bug reports*, which is mostly done by attaching discussions to a report.

Content of Bug Reports. Many communities provide detailed information on what should be described in a bug report. Still, the Arch community may best summarize what is relevant:

Arch
"If you do not know what the relevant pieces of information are, do not be shy: it is better to give more information than needed than not enough."
https://wiki.archlinux.org/index.php/Bug_reporting_guidelines

Besides this recommendation, we could find several codes from various distributions to define the content of a bug report. Most important is a summarizing *title* for searching and understanding the report. In addition, the report should describe *observed and expected behavior*, enriched with *logged data* (e.g., error logs), and *screenshots or videos* to depict graphical bugs. To facilitate bug fixing, most communities ask to specify the *environment* (e.g., system configuration) and provide a step-by-step *replication* guide. Finally, some communities ask to *classify or tag* bugs (e.g., severity level, bug type, packages) and provide *contact data*.

We could further identify themes on how to write this content and what not to put into a report. Namely, this includes some *soft-skills* related information, stating that a bug report should be polite and not include gossip. Bug reports should also not comprise *sensitive data*, such as passwords. While *preparing* a bug report, most communities ask the contributor to first check whether it already exists or has been fixed (e.g., in a newer or nightly version). Some communities also codify whether to report something or not:

Lubuntu
"MOST IMPORTANTLY: if you're not sure if you should report a bug, report it anyway!"
https://phab.lubuntu.me/w/bugs/

Important for most communities is to post only a single issue in each bug report. The codified information we found highlights that most communities mostly care about getting informed about bugs. They seem to prefer that as much information as possible is provided, that all potential bugs are reported, and that they do not have to separate multiple bugs tangled in a single report.

Discussion. The details of the information codified on quality assurance vary heavily, sometimes only specifying where to submit a bug, while some communities define templates. Some communities report processes to determine whether a bug is a bug, whether it has been reported, and what information should be provided. For example, Mint provides a detailed introduction on what general concepts or steps users should be aware of, including observation, expectation, reproducibility, responsibility, change, errors, environment as well as steps for reporting bugs.[10] Unfortunately, such detailed

[6] https://docs.fedoraproject.org/en-US/project/code-of-conduct/
[7] https://fedoraproject.org/wiki/Join_the_package_collection_maintainers
[8] https://www.deepin.org/en/developer-community/development/
[9] https://dev.getsol.us/
[10] https://linuxmint-troubleshooting-guide.readthedocs.io/en/latest/

guides are rare, but may provide a basis for other projects, guidelines, issue trackers, and analyses to improve practices. Detailed information in issue trackers may help advancing techniques, for example, to assign bugs to issue trackers, evaluate testing techniques, classify bugs or automated data collection and report creation.

RQ$_4$: What bug-reporting practices are publicly codified?

We found that mainly information on bug reporting is codified, but many details about the quality-assurance practices employed remain unclear for externals.

8 THREATS TO VALIDITY

Internal Validity. The documents we analyzed were produced by the community of a distribution and may not represent their current practices. For example, documents may be incomplete, outdated, inaccessible or wrong (cf. Section 4). So, we cannot conclude to what extent the communities employ practices, but we purposefully took this perspective of externals facing potentially tainted information.

We manually identified and analyzed natural-language documents. While we carefully read the documents and recorded our process, the outcome may be influenced by our opinions and knowledge. Moreover, we may have made errors or used varying depths while searching for documents, potentially resulting in false classifications or missed information. We aimed to mitigate this threat by using open coding and open-card sorting, by synchronizing our results and processes constantly, and by cross-checking the results.
External Validity. We analyzed Unix-like distributions, which essentially comprise the adaptation and maintenance of packages. These distributions are complex and require substantial knowledge, wherefore contributors have to undergo an intensive onboarding. We cannot determine whether other OSS/FLOSS projects are more open or codify more information publicly. Moreover, we did not analyze to what extent the provided information is actually useful to support onboarding. So, we cannot generalize our results, but our insights on practices and information provisioning are still highly valuable for communities and researchers.

9 CONCLUSION

In this paper, we qualitatively analyzed the publicly codified information on how to contribute to 25 Unix-like distributions. We investigated four research questions on how and what information is provided, identifying considerable differences between communities. Overall, our results are valuable for open-source communities to design and codify their project practices. For new developers and researchers, we highlighted challenges of identifying codified information about such communities. In future work, we intend to compare the codified information to the practices employed, We also aim to conduct interviews and surveys to understand what practices work better in what scenarios.
Acknowledgments. This research has been supported by the German Research Foundation project EXPLANT (SA 465/49-3).

REFERENCES

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *ASE*.
[2] Mark Aberdour. 2007. Achieving Quality in Open-Source Software. *IEEE Software* 24, 1 (2007).
[3] Sogol Balali, Igor Steinmacher, Umayal Annamalai, Anita Sarma, and Marco Gerosa. 2018. Newcomers' Barriers. . . Is That All? An Analysis of Mentors' and Newcomers' Barriers in OSS Projects. *CSCW*.
[4] Anton Barua, Stephen Thomas, and Ahmed Hassan. 2014. What are Developers Talking About? An Analysis of Topics and Trends in Stack Overflow. *Empirical Software Engineering* 19, 3 (2014).
[5] Glenn Bowen. 2009. Document Analysis as a Qualitative Research Method. *Qualitative Research Journal* 9, 2 (2009).
[6] Jailton Coelho and Marco Valente. 2017. Why Modern Open Source Projects Fail. In *ESEC/FSE*.
[7] Domenico Cotroneo, Roberto Natella, Roberto Pietrantuono, and Stefano Russo. 2010. Software Aging Analysis of the Linux Operating System. In *ISSRE*. IEEE.
[8] Kevin Crowston, Hala Annabi, and James Howison. 2003. Defining Open Source Software Project Success. In *ICIS*.
[9] Uwe Flick. 2018. *An Introduction to Qualitative Research*.
[10] Denae Ford, Justin Smith, Philip Guo, and Chris Parnin. 2016. Paradise Unplugged: Identifying Barriers for Female Participation on Stack Overflow. In *FSE*.
[11] Andrea Hemetsberger and Christian Reinhardt. 2006. Learning and Knowledge-Building in Open-Source Communities: A Social-experiential Approach. *Management Learning* 37, 2 (2006).
[12] Guido Hertel, Sven Niedner, and Stefanie Herrmann. 2003. Motivation of Software Developers in Open Source Projects: An Internet-Based Survey of Contributors to the Linux Kernel. *Research Policy* 32, 7 (2003).
[13] Ayelet Israeli and Dror Feitelson. 2010. The Linux Kernel as a Case Study in Software Evolution. *Journal of Systems and Software* 83, 3 (2010).
[14] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in Software Product Lines. In *ICSE*.
[15] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. 2019. Where is my Feature and What is it About? A Case Study on Recovering Feature Facets. *Journal of Systems and Software* 152 (2019).
[16] Jacob Krüger. 2019. Are You Talking About Software Product Lines? An Analysis of Developer Communities. In *VaMoS*.
[17] Jacob Krüger. 2019. Tackling Knowledge Needs during Software Evolution. In *ESEC/FSE*.
[18] Jacob Krüger, Jens Wiemann, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2018. Do You Remember This Source Code?. In *ICSE*.
[19] Simone Livieri, Yoshiki Higo, Makoto Matsushita, and Katsuro Inoue. 2007. Analysis of the Linux Kernel Evolution Using Code Clone Coverage. In *MSR*.
[20] Christopher Mendez, Hema Padala, Zoe Steine-Hanson, Claudia Hilderbrand, Amber Horvath, Charles Hill, Logan Simpson, Nupoor Patil, Anita Sarma, and Margaret Burnett. 2018. Open Source Barriers to Entry, Revisited: A Sociotechnical Perspective. In *ICSE*.
[21] Courtney Miller, David Widder, Christian Kästner, and Bogdan Vasilescu. 2019. Why Do People Give Up FLOSSing? A Study of Contributor Disengagement in Open Source. In *OSS*.
[22] Rohan Padhye, Senthil Mani, and Vibha Sinha. 2014. A Study of External Community Contribution to Open-Source Projects on GitHub. In *MSR*.
[23] Huilian Qiu, Alexander Nolte, Anita Brown, Alexander Serebrenik, and Bogdan Vasilescu. 2019. Going Farther Together: The Impact of Social Capital on Sustained Participation in Open Source. In *ICSE*.
[24] Jeffrey Roberts, Il-Horn Hann, and Sandra Slaughter. 2006. Understanding the Motivations, Participation, and Performance of Open Source Software Developers: A Longitudinal Study of the Apache Projects. *Management Science* 52, 7 (2006).
[25] Oscar Rodríguez, Ana Martínez, Jesús Favela, Aurora Vizcaíno, and Mario Piattini. 2004. Understanding and Supporting Knowledge Flows in a Community of Software Developers. In *Groupware: Design, Implementation, and Use*.
[26] Walt Scacchi, Joseph Feller, Brian Fitzgerald, Scott Hissam, and Karim Lakhani. 2006. Understanding Free/Open Source Software Development Processes. *Software Process: Improvement and Practice* 11, 2 (2006), 95–105.
[27] Ivan Srba and Maria Bielikova. 2016. A Comprehensive Survey and Classification of Approaches for Community Question Answering. *ACM Transactions on the Web* 10, 3 (2016).
[28] Igor Steinmacher, Tayana Conte, Marco Gerosa, and David Redmiles. 2015. Social Barriers faced by Newcomers Placing their first Contribution in Open Source Software Projects. In *CSCW*.
[29] Igor Steinmacher, Marco Silva, Marco Gerosa, and David Redmiles. 2015. A Systematic Literature Review on the Barriers faced by Newcomers to Open Source Software Projects. *Information and Software Technology* 59 (2015).
[30] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *CCS*.
[31] Josh Terrell, Andrew Kofink, Justin Middleton, Clarissa Rainear, Emerson Murphy-Hill, Chris Parnin, and Jon Stallings. 2017. Gender Differences and Bias in Open Source: Pull Request Acceptance of Women versus Men. *PeerJ Computer Science* 3 (2017).
[32] Bogdan Vasilescu, Alexander Serebrenik, Prem Devanbu, and Vladimir Filkov. 2014. How Social Q&A Sites Are Changing Knowledge Sharing in Open Source Software Communities. In *CSCW*.
[33] Thomas Zimmermann. 2016. Perspectives on Data Science for Software Engineering. Chapter Card-Sorting: From Text to Themes.