Socio-Technical Distillation

From Micro-Level Responsibilities to Macro-Level Architecture Views

Pia Eichinger
Matriculation Number: 3267895
Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science (M.Sc.)

Master Thesis

Faculty of Computer Science and Mathematics
Laboratory for Digitalisation

Date of Submission: September 30th, 2022
Supervisor: Prof. Dr. rer. nat. Wolfgang Mauerer
Co-Advisor: Dr.-Ing. Ralf Ramsauer

Abstract

The development of large software systems requires thousands of individuals to collaborate. This necessitates a logical decomposition of the system into smaller, manageable pieces, augmented by clearly defined ways of appraising and admitting modifications to the code base. While software architectures and integration processes are established means, neither can be automatically inferred from fundamental technical artefacts, such as source code. Rather, they require a-priori human involvement, judgement, and abstraction. Yet commonly, maintaining the formal description of architectures and process specifications is not a primary concern.

I show that often, open-source projects already contain well-tended micro-level information on code responsibility, and therefore the required human knowledge. In this work, I automatically derive macro-level views of software architectures, enriched with semantically understandable component identifiers without direct human involvement.

In this work, I show how to visually track the temporal evolution of the derived macro-level architectural views. I argue that my results form a basis for quantitatively judging quality properties of projects. This is exemplified by applying my methodology to a specific use case, where I assess component viability for safety-critical software and other semi-formal certifications.

Finally, I evaluate my methodology using a carefully crafted mixed-method approach, comprising statistical modelling and analysis, expert-based assessment of results, and targeted interviews with key developers.

Zusammenfassung

Die Entwicklung großer Softwaresysteme benötigt die Kooperation von tausenden individuellen Entwicklern. Dies bedarf einer logischen Zersetzung des Systems in kleinere, handhabbare Teile, erweitert durch klar definierte Wege, um Modifikationen an Code zu bemessen und zuzulassen. Während Softwarearchitekturen und Integrationsprozesse etablierte Mittel sind, kann keins von beiden automatisch aus technischen Artefakten wie Source Code erschlossen werden. Stattdessen benötigen sie a-priori menschliche Involvierung, Ansicht und Abstraktion. Jedoch ist die Instandhaltung formaler Beschreibungen von Architekturen und Prozessspezifikationen oft kein primäres Anliegen.

Ich zeige, dass open-source Projekte häufig bereits gut gepflegte micro-level Informationen über Code-Verantwortlichkeiten besitzen und somit also das vorausgesetzte menschliche Wissen. In dieser Arbeit leite ich automatisch macro-level-Sichten auf Softwarearchitekturen her, bereichert mit semantisch verständlichen Komponenten-Identifikationen ohne direkte menschliche Mitwirkung.

Im Rahmen dieser Arbeit zeige ich, wie man die zeitliche Entwicklung der abgeleiteten macro-level-Sichten auf Architekturen verfolgen kann. Ich argumentiere, dass meine Resultate eine Basis für quantitative Bewertung von Projektqualitätseigenschaften formen. Dies veranschauliche ich durch Anwenden meiner Methodik auf einen spezifischen Anwendungsfall, in dem ich Komponentenrealisierbarkeit für sicherheitskritische Software und andere semi-formale Zertifizierungen beurteile.

Letzendlich evaluiere ich die vorgestellte Methodik mithilfe einer sorgfältig gefertigten und gemischten Validierungsmethode, die sich aus statistischer Modellierung und Analyse, expertenbasierter Resultatevaluierung und gezielten Befragungen mit zentralen Entwicklern zusammensetzt.

Contents

1. Introduction and Contributions

Large software systems (*e.g.*, the Linux kernel) comprise millions of lines of code, oftentimes created by tens of thousands of developers. Code quality rests on several pillars—proper use of implementation language and tools, an architecture that provides a high-level structure, and processes for evolution management [4].

Since it is impossible for a single individual to curate the entire software system, code bases are often portioned into manageable areas of responsibility [16, 76]. In Open-Source Software (OSS) projects, developers responsible for a particular area are usually called *maintainers* [32]. As domain experts, they are in charge of reviewing, shepherding and integrating code changes (*i.e.*, patches) proposed by contributing developers. Maintenance and responsibility strategies are firmly rooted in OSS development [52, 104, 29]: Code is peer-reviewed and validated by other knowledgeable individuals, and then integrated by maintainers [4, 76, 31, 32]. Such strategies are supposed to ensure high code quality and avoid common mistakes up front [4, 1]. Similar approaches are employed in commercial settings [97]. In particular, system-level components, which will be subject for analysis in this work, predominantly use mailing lists for developer and code discussion, even if other projects have started to employ different means [49].

Software undergoes continuous change, with an ever-shifting workforce [9], triggering architectural drift, away from the initial specifications, which are often not kept in sync with the factual state of the code base [3, 34, 50, 103]. Likewise, the documented processes usually fail to faithfully describe the *de-facto* approach of communities. Often, architecture and processes may not even be documented in the first place. Although distilling software architectures from the code base has been intensely studied [62, 58, 6, 7, 94, 91, 57, 5, 40], it remains, in the words of Bass, Clements and Kazman [13], an "interpretative, interactive, and iterative process [...] that it is not automatic".

It is based on *both*, technical *and* social aspects. Hall *et al.* [40] state that, as of 2018, "existing remodularisation algorithms have sought to produce improved designs automatically, but have been unable to do so satisfactorily", and that they "must necessarily involve a degree of input from an expert".

Multi-Step Approach. I propose a multi-stepped approach towards distilling macro-level architecture views from socio-technical artefacts, as also visualised in the description of my data analysis pipeline in Figure 1:

① *Methodology:* We can learn from existing engineering artefacts that *implicitly* capture information on social aspects, and use these low-level (micro) artefacts to distil a high-level (macro) architecture view with semantic explainability, in a fully automatic way.

② *Use case:* While my methodology is largely generic, I demonstrate its applicability to one important use case: how to ensure process conformance and architectural cohesion of software modifications. This is particularly relevant for the composition of safety-critical systems based on OSS components, as many safety standards take adherence to a-priori defined processes as major quality criterion, which is however in stark contrast to OSS development that is self-organised and without means of centrally enforcing process policies. Based on my techniques, I suggest and evaluate a novel approach that quantitatively measures such conformance in an ex-post setting. This could open a future bridge between safety-critical development and high-quality open-source components, because sufficient quantitative evidence for their qualities could then be provided by my approach. These could be appreciated by safety standards, despite the fact the integrating a-priori prescriptions into OSS development processes would still remain impossible.

③ *Mixed-method validation:* To validate both the generic methodology and the specific use case, I perform a carefully designed mixed-method analysis [30, 64], using state-of-the-art sta-

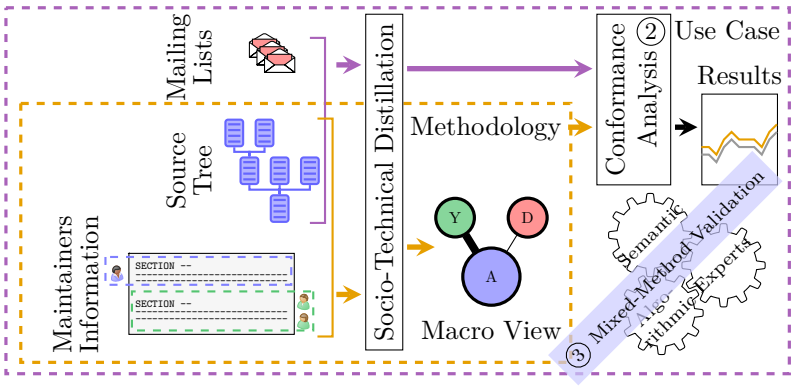


Figure 1: Analysis pipeline to implement my approach. ① Methodology: Distillation from micro- to macro-level view. ② Use case: Conformance analysis. ③ The extensive mixed-methods validation of my results.

tistical methods, expert-based classification, and maintainer interviews. These validations are directly appended after the introduction and results from both the methodology and the use cases.

Areas of responsibility. In software development, social responsibilities for technical artefacts are captured by a semi-formal, yet machine-readable mapping between code artefacts and developers, such as `MAINTAINERS` files. It connects parts of the source tree to areas of responsibility, the *sections* or *subsystems* [25]. In turn, these are assigned to the care of one or more human maintainers.

Architectural views in the software engineering literature are often based on system components [40, 58, 62]. This point of view differs from structures declared in the `MAINTAINERS` file, as the latter can provide several thousand areas of responsibility, drastically more than than the tens of architectural subsystems typically considered for large software systems like the Linux kernel [59]. While high-level architectural views are based on *macro-level* subsystems, responsibility sections provide a *micro-level* view.

As I show in this work, areas of responsibility can be a basis for deriving information closely related to architectural decomposition, and can serve to uncover effective reference processes. In other words, I argue that the macroscopic architecture and process view can be derived form a microscopic assignment of responsibilities. Given that OSS projects often provide no or only outdated architectural information [42], this is important to derive notions of processes that are directly rooted in systems themselves, without having to rely on any externally mandated specification. As the change history in maintainers specifications shows (see below for a quantitative analysis), developers take great care to keep the information up-to-date, which ensures that derived macro-level views are aligned with changing code bases.

Contributions. In my approach of establishing a methodology, evaluating it on a specific use case, and submitting both to a mixed-method validation, my contributions are the following:

- Based on structural properties of the micro-level view, as well as social network analysis, we distil architectural macro-level views. Different from earlier approaches integrating sociological knowledge, my analysis is fully automated.
- As part of my methodology, I establish a visualisation of the resulting structures which is

interpretable, since all components are meaningfully labelled. I am further able to visually demonstrate their evolution over time.
- As part of my use case, I trace the flow of patches from their initial conception, via discussion and refinement, to eventual integration, and compare the effective integration paths with the prescriptions derived from the views.
- As part of my mixed-method validation, I confirm, based on expert knowledge and developer interviews, that the macro-level architecture view represents a meaningful decomposition of the studied system.
- In validation, I also define measures to quantify the degree of conformity to self-prescribed integration processes, and study their temporal evolution for the four subject projects Linux, QEMU, U-Boot, and Xen, characterised in Table 2.

Structure. I first give an overview of related work in in Section 2. I furthermore present all mathematical concepts and algorithms that my method, my analysis or my validation employs in Section 3. In Section 4, I explain the technical preliminaries that my work analyses and uses. The tool PaStA, the central tool for all my analyses, is presented in Section 5. In Section 6, I explain the micro-level- and macro-level view on software systems, and describe my methodology. I then present a concrete use case that investigates process conform integration of code changes in Section 7. Next, I elaborate on the reproducibility of my work and results in Section 8. The discussion in Section 9 elaborates on possible threats to the validity of the work. Finally, Section 10 concludes this thesis and gives an outlook on future work.

Reproducibility. The analysis pipeline is available as open-source software on the supplementary website (link in PDF) and is accompanied by a fully automated reproduction package. This includes all raw input data used in calculations, a deployment of the analysis software (self-contained, without dependencies on external components), as well as all post-processing scripts to evaluate and visualise the data.

Credit. This work, specifically the chapters 2, 4, 6, 7, 9 and 10 all share material with a pre-printed paper "Socio-Technical Distillation: From Micro-Level Responsibilities to Macro-Level Architecture Views", authored by Pia Eichinger, Ralf Ramsauer, Christian Hechtl, Thomas Bock, Sven Apel, Stefanie Scherzinger and Wolfgang Mauerer.

2. Related Work

Analysing code responsibility registries. The role of a maintainer with a designated area of responsibility is established in software development, as well as the custom to discuss patches on mailing lists [76]. Specifically, my work relies on analysing MAINTAINERS files which are established artefacts in empirical software engineering research, and commonly used to identify which developers are also maintainers [32, 54, 26]. Further studies observed, among other aspects, the workload distribution among maintainers [104, 84, 85, 71]. Similar analyses have also been conducted among practitioners [25]. However, I am not aware of attempts to derive macro-level architectural views from MAINTAINERS.

A related concept are the Google [66, 97] `OWNERS` files. At least one code owner *must* participate in the code review process, which is a stricter requirement than for MAINTAINERS based processes.

GitHub further introduced `CODEOWNERS` files [39] in 2017, which are by now well-adopted.[1] Different from MAINTAINERS, sections in `CODEOWNERS` are not explicitly declared with titles, but often, projects have the convention of diligently assigning section titles within comments, de-facto very similar to MAINTAINERS. GitHub has built-in support for `CODEOWNERS` format, such as automatically suggesting suitable maintainers for code review.

Identifying subsystems. Software architecture and automated system abstraction is related to this work, and heavily researched. Consequently, I can only highlight selected seminal works.

Automated solutions for reverse engineering software from source code date as far back as 1993, when Müller *et al.* presented (semi-)automated solutions, using graphs and clustering. They have been intensively studied thereafter [94, 6, 7, 57], a highly prominent example being the tool Bunch by Mancoridis, Mitchell *et al.* in 1999 [58, 60].

Though many approaches inspecting the code base (or, in the work of Beyer *et al.*, code changes [15]) have been implemented over the years, with or without manual domain knowledge input, it is still a heavily researched topic since current solutions wield unsatisfactory results, as Hall *et al.* argue in their work on the SUMO tool [40]. Similar to my approach, Hall *et al.* recast their notion of software modularisation as a set partitioning problem. My approach differs insofar as mine is completely automated and decoupled from code semantic, whereas SUMO requires corrective feedback from developers.

The need for incorporating social responsibility artefacts is established. In 1999, Bowman *et al.* presented a case study on Linux that showed that understanding large software architectures through social artefacts has potential: "We suggest that all ownership architecture that documents the relationship between developers and source code is a valuable aid in understanding large software systems" [19]. They discuss that "ownership architectures identify experts for system components, show non-functional dependencies, and provide quality estimates for components. This information is useful for understanding systems."

Their work groups developers and maintainers into social communities. In contrast, my work leverages information of the MAINTAINERS files, which assigns source code artefacts to social responsibilities, together with informal descriptions of the particular areas. In [9], Ashraf *et al.* showed that social communities can significantly change over time. Since my method is fully automated, and relies on curated input artefacts, it can deal with such changes.

In 2005, Andritsos *et al.* cites Bowman's *et al.* work as having shown the high potential of using ownership information in fully-automated reverse engineering, but states that its merit is yet to be evaluated. Rather than using responsibility information, they introduce a hierarchical clustering

[1] Proliferation of `CODEOWNERS` reaches over 7.5k OSS GitHub projects at the time of this writing, as observed with a BigQuery search.

algorithm of a software system based on information loss for files and its dependencies [5]. As of 2022, Bowman's work dates far back, but has (to the best of the my knowledge) not been applied and verified yet, which makes my method the first to do so.

Common applications of derived abstractions are system comprehension [40], or detecting violations of coding patterns. For the latter, I refer to the comprehensive book chapter by Lindig [56]. In contrast, I will introduce a new use case that has not yet been studied in this context, namely the process conformance in integrating patches into the code base.

Patch analysis. Previous work on patch integration focuses on the speed of integration, or prediction thereof [48]. Yet in my use case, I will target the correctness of patch integration.

Certain technical aspects of my work defer to the open-source tool PaStA [72, 73], originally designed to detect semantically similar patches on mailing lists and in revision control systems.

Social-network analysis. To detect structures within graphs, there is a family of algorithms for Social Network Analysis that arrange multiple disjoint subsets of nodes into cells. Popular representatives of this family the random walk clustering methods, where the underlying idea is to randomly "walk" along edges. A random walker will spend considerable time within a community, due to the higher edge density [33]. Specific implementations (such as [27]) extend the walktrap algorithm by Pons and Latap [65] to also consider edge weights.

3. Mathematical Background

The research presented in this thesis relies on various mathematical concepts. To provide insight into the theoretical background of this work, I will explain them in this chapter.

3.1. Community Detection in Graphs

Graphs are one of the most broadly used mathematical concepts to represent things such as a system or a network. Its simplicity allows for easy modelling and analysing of structures. To detect certain partitions in a graph, so called *communities*, there exist various community detection algorithms. A desired property of this partitioning is a high edge density among its vertices [33].

Proper and statisfactory community detection is still very hard despite multiple approaches that were developed and researched over the years [33]. Since the "correct" way of grouping a graph can be a very opinionated topic, I used various community detection algorithms, namely Walktrap, Louvain and Infomap, to compare to each other and validate my results. A visual comparison of the four algorithms can be seen in Figure 2. We will briefly present and explain all four algorithms.

Walktrap. The first method is called Walktrap. It was developed by Pons and Latapy [65]. The underlying idea is that of a walker, traversing a graph while choosing edges at random, getting "trapped" within a group of densely connected vertices or spending more time among them. These groups will then be considered the communities of the graph..

The algorithm by Pons and Latapy uses a walker with random walks of length t, becoming "trapped" in densely connected clusters. The originally proposed method does not consider edge weights, only the presence or absence of edges, but can be easily extended to do so [65].

As implementation of the algorithm I use `cluster_walktrap` provided by the igraph package [27, 17]. This implementation extends the original algorithm to take edge weights into account. Edges with higher weights are more likely to be chosen for the random walk, making vertices with not only dense but also highly weighted edges more likely to be considered a community.

A single discrete random walk process has a walker "sitting" on a vertex and randomly choosing the next destination vertex among the directly connected nodes, i.e. among the *neighbourhood* of the vertex. A distance between the vertices is later calculated to determine which belong in the same community.

Infomap. Rosval *et al.* proposed another community detection algorithm, commonly referred to as "infomap" [78]. It is designed to reveal community structure in weighted and directed networks. The network is composed into various modules or communities by compressing a description of information flow on it.

The key idea to their algorithm is to express the networks data streams as code, which is then used to efficiently describe a random walker on the network. Finding the community structures is then equivalent to solving a coding problem.

Fast-Greedy. Clauset *et al.* developed another commonly used community detection algorithm. It is designed to wield suitable results for very large networks with acceptable computational costs and is usually referred to as "Fastgreedy". It starts with a division of the network based on subnets of vertices that are already highly connected and iteratively improves the subnets by adding edges [24].

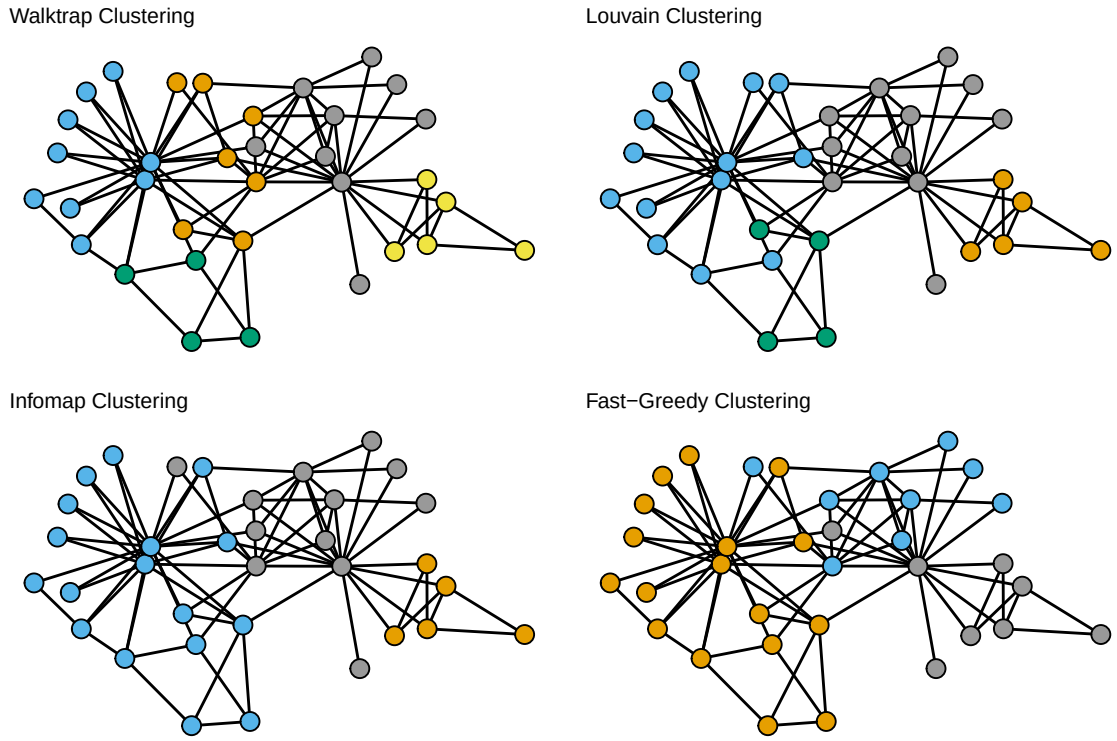


Figure 2: The same graph with coloured communities detected by four different clustering algorithms. It uses the Zachary Karate Club network example from igraph [28, 101].

It relies on a quantity of networks commonly referred to as *modularity* [63], which measures how easily a graph can be divided into numerous communities.

Louvain. Another community detection algorithm was proposed by Blondel *et al.* [17]. It uses modularity optimization to achieve high modularity and therefore a good clustering solution.

3.2. Normalised Compression Distance

Normalised Compression Distance (NCD) is a method to measure similarity between data points based on a data compressor. It is a non-negative number $0 \leq r \leq 1 + \epsilon$ representing how different two data points are. The smaller, the more similar. The ϵ accounts for imperfections in compression techniques, though these are very unlikely [23].

It is known for its general applicability, noise resistance and was shown to be theoretically optimal. It can, free of parameters, compute distances between arbitrary data vectors.

The NDC approximizes the *Normalised Information Distance* which relies on the notion of the Kolmogorov Complexity. The underlying idea for NDC is as follows: by using a compression algorithm on individual data vectors and their concatenated results, I can measure how distant they are [11].

3.3. Repository Mining

Code repositories or mailing lists are examples of publically available archives of software projects and software development. The research field for mining software repositories (MSR) analyses the data of these data treasure groves to unveil interesting new insights into development processes or the software systems themselves [43].

In this work, I will make use of a repository mining tool to quantify and analyse projects.

3.4. Cluster Similarity

We have employed multiple algorithms to detect clusters in a network, which I all discussed and presented before. However, the choice of the "true" algorithm remains subjective. Any clustering that results from any algorithm can be prone to errors and misclassifications and if the right way to cluster a network could be decided in an objective manner, it would not remain such a heavily discussed research topic.

To verify the suitability of my method and show that my results wields stable outputs across all clustering algorithms, I made use of two different measures to evaluate cluster similarity: Purity [102] and V-measure [77]. The measures effectively express how well an existing clustering compares to the "true" clustering, a ground-truth. Since I do not have a ground-truth in my method, I simply compare the clusterings against each other to view how similar they are.

3.4.1. Purity

Purity measure expresses to which extent clusters contain primarily elements from one single class. Given a singular cluster S_r of size n_r, the purity of the cluster is defined as:

$$P(S_r) = \frac{1}{n_r} \max_i (n_r^i)$$

This is the fraction of the overall cluster size to number of the largest singular class in that cluster. The overall purity measure of a clustering is given by a weighted sum of all cluster purities as follows:

$$Purity = \sum_{r=1}^{k} \frac{n_r}{n} P(S_r)$$

The larger the purity value, the better the clustering solution. However, it is biased against large clusters for which it intrinsically produces good results.

3.4.2. V-Measure

Along with purity, entropy is also a commonly used measure for assessing clustering quality. However, similar as to purity, entropy only measures how only data points of a single class are assigned to a single cluster. This property is also commonly referred to as *homogeneity* [77]. While homogeneity is a desireable property of good clustering quality and states how well a clustering contains only elements of a single class, it does not touch on whether the clustering also contains *all* elements of a class, commonly referred to as *completeness*. V-Measure combines both completeness and homogeneity by measuring if a clustering solution contains all and only data points of a single class.

4. Technical Preliminaries and Definitions

In this chapter, I will explain the core technical preliminaries for open-source software and its development processes. We will also introduce the open-source projects that were analysed in this context.

4.1. Open Source Software

The definition of Open-Source Software (OSS) oftentimes varies in common literature and is subject to numerous discussions [36]. In the words of Wang *et al.*, "we do not have a universally accepted definition of OSS" [96].

According to Fuggetta [36] the obvious meaning for OSS is that the source code is published and can be viewed by anyone. The author argues that this is a very simple definition and not the meaning that its advocates intend, which is more in the sense of "free software". Fuggetta describes free software as "the users' freedom to run, copy, distribute, study, change and improve software."

Another, more verbose, definition is provided by the Open Source Initiative [90]. They state that access to the source code is not enough to qualify as OSS. The distribution of OSS has to comply with the following criteria.

Free Redistribution The software can be sold and given away as a component of aggregated software. No royalty or fee is required for this type of redistribution.

Source Code The source code must be freely available and in a readable format. Deliberately obscure source code, which hinders understanding, is not allowed. The distribution must be allowed in source code as well as compiled form. If the compiled form is distributed for whatever reason, the obtainment of the source code must be made accessible with little to no extra charge, for example by downloading the source code from a link.

Derived Works Derived works and modifications must be allowed under the same distribution terms as the original software.

Integrity of The Author's Source Code Restriction of the distribution of the modified software may be allowed only if the distribution of the software with patches, modifying the software at build time, is allowed.

No Discrimination Against Persons or Groups No single person or group of persons can be discriminated against for using the software.

No Discrimination Against Fields of Endeavor The software can be used in any field, for example in businesses or in research.

Distribution of License The restriction of only letting the software be distributed as a part of particular software distribution. The software itself

License Must Not Restrict Other Software Restrictions on other software distributed alongside can not be paced.

Table 1: Quantitative characteristics of the subject projects. It compares both the projects' amount of maintainers and the total Lines of Code, measured in units of thousand, from start to end of the analyses' time window.

	$t_1 - t_0$	#Maintainers		kLoC	
Project		t_0	t_1	t_0	t_1
Linux	11 years	712	1618	14533	32233
QEMU	10 years	32	157	801	3090
U-Boot	8 years	38	94	1890	3768
Xen	11 years	28	34	877	886

Licence Must Be Technology-Neutral The provision of the software can not depend on specific technology.

Software Licenses which comply with these criteria qualify for OSS.

Despite often relying on volunteer contributions, openly developed software has a reputation for exceptionally high quality and various other benefits, such as:

- **Public Code:** Since the code of OSS is public, anyone interested can view the code. This often results in volunteers reviewing, extending and enhancing the code.
- **Higher Security:** Despite the open accesibility of OSS, it has been proven to be more secure. This is a result of the volunteering community and experts who are detecting exploits and contributing fixes to the Software [79].
- **Fewer Costs:** The free contributions by volunteering developers lower the overall development and long-term maintenance of software [79].
- **Higher Reliability:** As cited by Raymond: "Given enough eyeballs, all bugs are shallow" [75], meaning that the exposure of code to a large community of reviewers will eventually detect all flaws, making them known and opening the door to fixing them.

4.2. Analysed Open Source Projects

I applied my methods to the four open-source subject projects as listed in Table 1, in which I present their essential characteristics. These analyses start at the point in time for which there is the first valid revision of a MAINTAINERS file. This allows me to consider around ten years of historical evolution. Since my use case deals with safety scenarios, I deliberately preferred analysing a smaller number of projects that have found initial use in safety-critical appliances to using a larger selection of projects that might never share such strong requirements. Thereby, I accept a small loss in statistical power. I also chose my subject projects such that they impact the low-level aspects of systems; the higher up an application is in the software stack, the more methods to ascertain safe operation become available (*e.g.*, virtualisation, containers, sandboxes, redundant computation, …), and the influence of software quality decreases. The projects will be introduced and briefly explained in the following sections.

4.2.1. Linux

The Linux Kernel is the largest and most well-known project. It is "a clone of the operating system Unix, written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net" [2].

Linux is used essentially everywhere, ranging from phone devices to spacecraft on-board software, and is widely known for its reliance and security. As an operating system, it is the interface software between software and hardware, managing things such as peripheral devices, CPU and memory. It has been developed with Git as the main revision control system since 2005. The current source code can be found in Linus Torvalds' git repositry [2]. Maintenance and development is done through an open development process. With millions of users the number of contributions has greatly increased [88, 55].

Due to its broad usage, an ongoing and successful development process that ensures and maintains its high standard of quality is in the best interest of many major companies that utilise its code. This resulted in a strong cooperation, by which companies such as IBM and Oracle actively contribute to the project [3].

Since Linux is known for its high standards of quality and reliance, there is interest in enabling it safety-critical systems. This interest sparked the launch of the project Enabling Linux in Safety Applications (ELISA) [87]. Linux was chosen and analysed due to its high demand in the safety-critical community and to contribute valuable information to ELISA.

4.2.2. QEMU

According to the official QEMU Wiki, "QEMU is a generic and open-source machine emulator and virtualizer" [69]. It can run operating systems or programs for one machine on different machines, such as a ordinary user PC, while achieving very good, or even near native, performance. It was developed by Fabrice Bellard in 2005 [14] and has since grown in size and influence. Its use cases vary from cloud testing to IoT development [89].

4.2.3. U-Boot

U-Boot is a boot loader for Embedded boards for processors like PowerPC, ARM, MIPS and several others. Its development and development process is closely related to that of Linux, going so far as to add extra support for Linux images [93].

It is the most common bootloader for Linux systems and supports many other embedded development boards and can be modified for specific hardware. It has lots of drivers, different filesystems and support for device trees. These features makes U-Boot a comfortable option for embedded systems.

4.2.4. Xen

According to the Xen-wiki, Xen is an open-source bare metal hypervisor, which is "a form of virtualisation in which the hypervisor runs directly on the underlying hardware" [38]. It can run various instances of the same or different operating systems in parallel on the same machine [99].

From the start of its development, it was designed for cloud computing and has since grown, having over 10 million users. Its community focuses on advancing virtualisation in both commercial and OSS applications. Its use cases range from server and desktop virtualisation to security, embedded and hardware appliances [98].

4.3. The Development Process

As all projects have an open development process, anyone is free to contribute code and patches to the repository. All projects use git as source code management tool, which offers built-in

[2] Available as OSS on https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/
[3] Patches from IBM and Oracle

functionality to format ones patches and commits to send it as an email.

To avoid the chaos of many developers formatting their patches and sending pull requests to the people in charge of the projects, creating a massive influx of unclear and complex code contributions for different purposes, each using their own unique coding style, it is necessary to put guidelines into place. This is especially important for projects as large as Linux.

All four projects use near identical requirements to their development process as Linux, with little to no variation. QEMU and U-Boot both reference the Linux conventions in their guides [68, 92], with U-Boot even going as far as to guide new users to Linux guidelines at the start of their own guidelines.

Since the development processes are extremely similar, I will provide a rough overview to convey the necessary concepts and technical preliminaries for this work.

4.3.1. Patch Implementation and Design

The guidelines on patch submission start as early as the design of the patch. For example, the Linux guidelines suggest involving the community as much as possible to prevent later redesigning [86] and the U-Boot guidelines recommend discussing ideas before implementing them [92]. If the entire community comes together to think of the proposed change, if it really is necessary and what consequences it could have, futile or duplicate work can be avoided early on. This holds especially true for new developers joining the community, since they might still lack the comprehension of large code bases and its multifaceted usages.

If the community deems a code change sensible, the developer can start implementing. According to all guidelines, the overall change is to be partitioned into singular logical changes, whereas one patch symbolises one independent modification. These patches are represented by git commits. Their commit messages need to follow specific formatting and with comprehensive descriptions. This is supposed to make reviewing easier and help people in the future who are trying to understand what the patch does and why it was necessary, e.g. for debugging purposes. Furthermore, the commits needs to be signed by the author. QEMU and Linux guidelines state that this is to legally express that the author agrees with this patch being contributed to the repository [68].

4.3.2. Peer Review and Patch Acceptance

If the implementation is finished and the patch coheres to the general submission guidelines, it is ready to be peer-reviewed, which is a highly valued practice in OSS development [52, 104, 29]. All projects are split into several thematical subsystems [16, 76], e.g. Networking for Linux. These split the code base into various manageable areas of responsibility which are maintained by so called *maintainers*. These are special developers who possess expert-level domain knowledge for their area of responsibility. They are in charge of reviewing and integrating proposed code changes by volunteering developers [32].

Mailing lists are widely used in OSS development and is also common practice for my analysed projects [73, 92, 83, 68]. For every large subsystem, there usually also exists a relevant mailing list where code changes for the subsystem are discussed, e.g. the networking support mailing list netdev@vger.kernel.org for all networking related patches. All projects also have a general mailing list where all patches - no matter its affected subsystem - are to be sent to.

The patch is then sent to all relevant mailing lists as well as the responsible maintainers for the affected areas of responsibility. Developers and maintainers can then discuss and comment on the change, suggesting modifications and giving general feedback. This can prevent major problems and bugs early on by having the author of the patch rework it through multiple revisions, if necessary.

In the end, the responsible maintainer has to accept the patch, which kicks off the integration into the repository. If the patch can withstand various rounds of testing and reviewing, the change is accepted into the official repository.

4.4. Maintainers Usage

As stated in the before, the code bases are usually split into various areas of responsibility - further called *sections* - which are maintained by developers with domain knowledge. Which maintainer is in charge of which files exactly is documented in a file called MAINTAINERS. If someone needs qualified input to specific code artefacts, the MAINTAINERS file directly assigns experts to the artefacts in question. This conveys a sense of ownership to code snippets and is helpful to finding the responsible people to send patches to.

In this section, I will introduce and elaborate on the MAINTAINERS file, its structure and its usage.

4.4.1. MAINTAINERS

MAINTAINERS is a file that is (by convention) located in the root directory of every considered project. It consists of a list of *sections* that provide a micro-level view on the repository. A section contains an informative description that sketches its content (*e.g.*, BUILDSYSTEM or LOGGING, *cf.* Figure 3), a list of assigned files and/or directories and the section's responsible maintainers. Additionally, a section contains information on public mailing lists where patches that affect the section should be sent to for discussion. In many OSS projects, especially the ones at the more fundamental layers of a software stack, code contributions are submitted to and discussed on mailing lists [73, 86, 49], which effectively implements public peer-review. Eventually, a patch is rejected, or integrated by a maintainer.

There is no formal standard for the structure of MAINTAINERS, yet most projects follow the Linux kernel conventions: starting with a short preamble of the project's development process, and followed by a so-called schema definition. I call an area of responsibility a *section* (also referred to as *subsystem* [25]). Beside other project-specific specifiers that only play a subordinate role for my analysis (*e.g.*, references to IRC chats, web pages, …), the relevant core components of a section are:

M: Maintainers (name, e-mail) responsible for this section,

L: Mailing list(s) that patches should be sent to,

F: Associated files and directories (with regex support), and

X: Wildcard pattern excluding files/directories.

Section overlap. A file or directory of the project may be assigned to multiple sections. Hence, sections may *overlap* in terms of shared source code. The overlap can be measured in standard metrics, such as file size or Lines of Code (LoC). Given the source code of a project, I can determine the pairwise overlap across sections. In Figure 3, sections `BUILDSYSTEM` and `LOGGING` share the file `scripts/debug.sh`.

In practice, MAINTAINERS files typically declare a section that always covers all files in the project (**F:** *), and which therefore fully overlaps with all other sections. Common names for such sections are "THE REST" for Linux, U-Boot, and Xen, or "General Project Administration" in QEMU. I refer to these sections as *catch-all section*, since their scope is the entire project. I will largely ignore them in laying out my methodology (Section 6), but they will play a role in presenting my use case (Section 7).

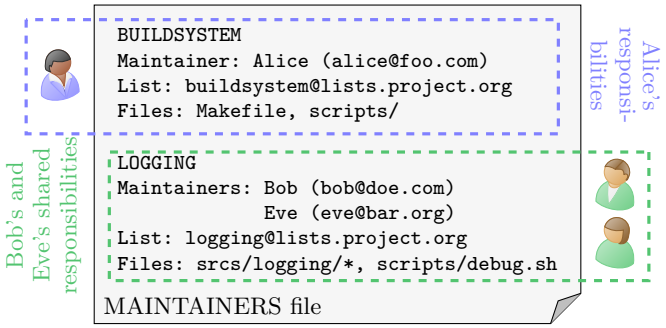


Figure 3: Structure of a MAINTAINERS file with different areas of responsibility (sketched).

Example 1. *Figure 3 shows a schematic MAINTAINERS file with two sections. The section titled* `LOGGING` *is maintained by Bob and Eve, whose e-mail addresses are are also listed. The relevant mailing list is also stated, and the files in the section scope are declared.*

Alice, Bob and Eve share responsibility for file `scripts/debug.sh`*. To restrict the responsibility of the file to the* `LOGGING` *section, a* **`X:`***-entry would have to be added to* `BUILDSYSTEM`*, showing that Alice is not in charge of* `scripts/debug.sh`*, despite maintaining everything else in directory* `scripts`*.*

We can measure the overlap of the two sections through the LoC or the file size of `scripts/debug.sh`*.*

4.4.2. The get_maintainer.pl Script

A new patch usually affects one or more sections, depending on which and how many files were changed. The people who are in charge of these files should then be notified. In the example of Figure 3, a change suggestion to `scripts/debug.sh` should be sent to all three maintainers in charge and the relevant lists for their sections.

Knowing where to send the patch is easy to determine in this very simplified example, but MAINTAINERS in practice can get very large, with projects like Linux spanning multiple thousand sections. Manually scanning the file and finding all relevent addressees would not be feasible for a productive development process. The various wildcard approaches on how to fine-tune relevance of a section for files further complicate this task.

The projects which use a MAINTAINERS file usually have their own script used for parsing the file to conveniently output the necessary data, such as `get_maintainer.pl` from Linux. They are put in place to provide assistance to developers who are seeking information. These usually take a file or a patch as input and output the relevant sections, maintainers and mailing lists, maybe some additional information depending on data from MAINTAINERS, therefore providing a convenient way to find out all addressees and additional necessary information for developers.

5. PaStA

Ramsauer et al. [72, 73] developed a tool to detect resembling patches: PaStA, the Patch Stack Analysis (PaStA) tool, which is available as open-source project, written in `Python3` and published under `GPL v2`. The method was implemented as an extension to PaStA while using its core functionality. This chapter will introduce the tool and explain how its infrastructure is beneficial to my use case.

5.1. PaStA Related Work

PaStA has been used for various other research topics. These have solidified the tool and its practices as valuable for research, all while establishing an infrastructure to analyse patches and development processes. We will give a short overview over some notable publications and results that have been achieved through PaStA.

Patch Stacks PaStA was initially developed to maintain sets of patches, often called "patch stacks". The tool can track the temporal evolution of patch stacks by mining git and determines patch similarity to measure integratibility for the core project. Its use was demonstrated as a use case on Preempt-RT [67], a real time extension of Linux [72].

Tracking of Commits As discussed before in Section 4, development for OSS projects is often done on public mailing lists. The patches are discussed, reviewed and reworked multiple times until the final version is committed and enters the repository. Keeping track of the multiple patch revisions is hard and not feasible to do manually.

PaStA is able to automatically and reliably link the mails on public mailing lists to their final version as a commit in the repository. The method was extensively tested and verified against a ground-truth [73].

Ignored Patches In the ideal case scenario, a patch is sent to a mailing list, reviewed, reworked and eventually integrated in its final version as a valuable extension to an OSS project. However, if a patch is neither answered, nor accepted by the developers and simply vanishes among the masses of the mailing list, it is considered ignored.

Duda *et al.* conducted a case study to determine how many patches were ignored in the Linux development, furthermore looking into the rate of ignored patches over time. To better understand the phenomon, they characterised the ignored patches to research possible discrimination overall or on certain subsystems or mailing lists taking place.

The characterisation of patch mails and their analyses was implemented as an extension to PaStA, creating an additional infrastructure to classify patch mails. We can later make use of this pre-existing infrastructure to further extend the characterisation of patches to determine patch conformance.

Mining Security Vulnerabilities While general development is done publically on mailing lists, Linux manages fixes to security issues - usually sensitive information which should not be broadcasted to a large audience - by having them notified to a small group of trusted maintainers. There, they are discussed and the fix is integrated without public involvement. The commit, despite being public, usually vanishes among the thousands of other commits for each release.

There are many projects that are based on Linux. The patches implemented in Linux are not always immediately pulled and integrated into these projects and security fixes are no exception.

This means that there is a significant time window between the moment security fixes appear as commits in the Linux repository and when they are integrated into dependent projects.

As earlier mentioned, PaStA is able to map patch mails to their eventual commit in the repository. By having these patches *not* appear on public mailing lists, PaStA can not map preceeding mails to the commit, making the commit without matching mails on public mailing lists.

With this knowledge, PaStA can mine exploits in Linux by searching for patches that did not appear on lists prior to integration. Ramsauer *et al.* detected 12 vulnerabilities in Linux within a time window of seven months with this method [74].

5.2. MAINTAINERS Parsing

To answer my research questions I require data that is generated by parsing the MAINTAINERS file. I need to produce a large data set containing information on responsible maintainers for every single patch integrated within my time window and affected sections for every single file in the repository. This would require running the script for a very large set of files and patches. Generating the necessary data for my analysis would require running the script for a very large set of files and patches.

While projects that use a `MAINTAINERS` file often come with their own parsing script (*e.g.*, `get_maintainer.pl` in the Linux project), they were not built to run in batch mode. For that purpose, PaStA has its own implemented MAINTAINERS parser logic. It can parse MAINTAINERS files in much less time, accelerating batch analysis.

Its implementation tries to mimic the behaviour of the original projects' scripts as accurately as possible. Along with the projects, the scripts and the MAINTAINERS files have been maintained and continuously changed across the different releases. Simulating the original script perfectly according to the time of integration is very difficult.

The original parsing script is defined as the ground-truth: a file or directory belongs to a section if the script states so and a patch was correctly integrated if the project's script declares this maintainer as relevant based on the corresponding version of MAINTAINERS. If we decide to only simulate the script, we might receive distorted outputs and therefore render any research results invalid.

In order to prove the validity of my data and results, I need to show that PaStA's implementation simulates each projects parsing logic well enough. I implemented a test script to compare both script outputs to address this threat to validity. This is further discussed in Section 9.

6. From Micro- to Macro-Level Views

This chapter outlines my methodology for socio-technical distillation.

6.1. Analysis Pipeline

Figure 1 provides a high-level overview over my analysis pipeline. For now, I focus on the part describing my methodology (label ①).

Lower left, I show the artefacts from which I first derive the micro-level, and then the macro-level view: this concerns the code repository, including the `MAINTAINERS` file, along with the entire development history. For parsing artefacts, I employ the third-party tool PaStA. I have substantially extended this tool, and made my patches, as well as my custom code (Python and R), openly available.

6.2. Micro-Level Views

Based on areas of responsibility (sections) declared in MAINTAINERS, I derive a *micro-level view* on the project, as visualised by the Venn-Diagram in Figure 4 (a). Each region corresponds to one section and is labelled with the section title. The region area corresponds to the lines of code associated with a section, and therefore its size. The areas shared by overlapping regions represent the extent of the code overlap between the respective sections. A natural assumption is that the more sections overlap, the stronger these sections are semantically related, which is what I base the micro-level view on.

Example 2. *Recall the example from Figure 3. It showed a simplified MAINTAINERS with two sections, overlapping with the script* `scripts/debug.sh`*. Showcasing this MAINTAINERS example in a similar manner as the Venn-Diagram in Figure 4 (a) would result in just two circles, labelled* `BUILDSYSTEM` *and* `LOGGING`*, both slightly overlapping. The overlap is caused by their shared file* `scripts/debug.sh`*. Their size and overlap would correspond to their size and the amount of shared LoC.*

6.2.1. Concept

My goal is to be able to analyse a MAINTAINERS, its sections and their relations of shared LoC among them. An intuitive way to depict entities and their relations is a graph, which leads to my first definition of a Micro-Level View.

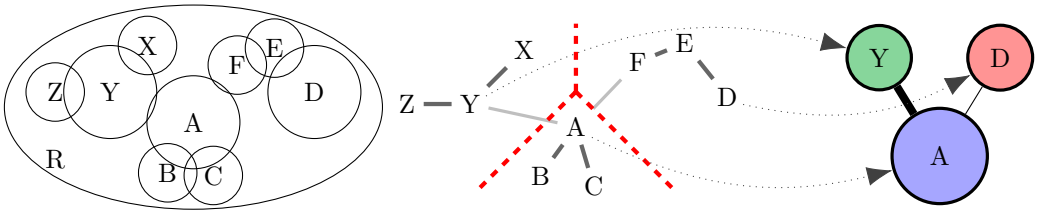
Definition (Micro-Level View)**.** *A Micro-Level View is an undirected graph.*

Given a section s, let LoC(s) return the total lines of code of all files in the scope of this section. We further generalise this metric to sets of sections in the natural way. Given a section s, title(s) returns the title of that section.

Given a snapshot of the code repository including a MAINTAINERS file, let S be the set of sections S. Each section is represented by one vertex in the graph. The vertex corresponding to a section $s \in S$ is labelled with title(s). The size of the vertex is defined as LoC(s).

There is an edge between two vertices v_1 and v_2 if there are sections s_1, s_2 that share LoC responsibilities in MAINTAINERS. Then, the edge is weighted by the total lines of code of all files within the overlap of s_1 and s_2.

Figure 4(b) shows an exemplary micro-level view graph. Given the sections on the left in (a) and their overlap, the corresponding micro-level view graph can be seen in (b). "Stronger" edges, i.e. edges with high weights, are displayed thicker than "weaker" edges. By doing this, I



(a) Venn diagram of section overlap (*micro-level view*).

(b) Abstracted *network view*, with sections partitioned into cells.

(c) Interpretable *macro-level view*.

Figure 4: (a) The MAINTAINERS file declares sections A, B, ..., F, and X,Y,Z. Given the code repository, we quantify section size and overlap in lines of code (b) Recasting the micro-level view as a socio-technical network: Overlaps are translated into edge weights (visualised by edge thickness); community detection partitions the sections into cells. The macro-level view (c) provides an interpretable visualisation, where node size and edge thickness indicate the extent of the scope and overlap, both quantified in lines of code. Vertices are labelled with the title of the largest section within the cell, fostering the interpretability.

can display how closely two sections are related. An exemplary micro-level view for Xen can be seen in Figure 5. Only the largest sections have their name displayed as a node label. Colour indicates, to which community the node belongs.

6.2.2. Validity

Under the assumption that MAINTAINERS approximately reflects a system's architecture based on shared code, the micro-level view accurately depicts these shared LoC relations, given my definition for the graph. The provided definition is very slim and intuitive and does not leave any room for interpretation or human involvement, since the entire view is automatically generated from MAINTAINERS.

A possible threat to the semantic validity of my graph is the choice of the ground-truth, the MAINTAINERS file, which I will address in Section 9. Another threat could be that while MAINTAINERS as ground-truth is a correct assumption, my choice of architecture, depicted by shared responsibilities, is wrong and creates misleading results.

MAINTAINERS is, by definition, a file for responsibility assignments. The responsibility is assigned to code artefacts, which can be measured. A threat that claims that responsibility assignments in MAINTAINERS are the wrong way to measure and depict MAINTAINERS accurately effectively claims that the file has a different purpose entirely, which contradicts its intended usage.

I deem these threats invalid to my micro-level view definition due to its simple structure which stays true to the principle of MAINTAINERS or ownership files in general.

6.3. Network View

In large projects, micro-level views exceed human cognitive capacities: Projects such as Linux comprise over 2,000 sections, which makes the micro-level view extremely overloaded and not readable. However, the composition of the resulting micro-level view wields interesting insights, especially when further inquired into even smaller scale substructures consisting of a subset of the total vertices. The network view is supposed to shine a light on particular substructures to analyse them.

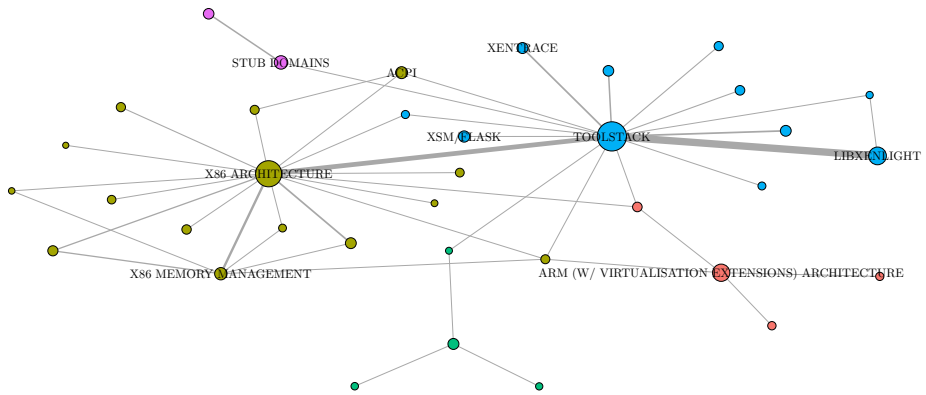


Figure 5: Xen version RELEASE-4.15.0 as micro-level view. Isolated nodes were removed to prevent visual clutter.

As a first abstraction to achieve a view for easier inspection, I recast the micro-level view as a (social) network. Figure 4(b) shows the corresponding undirected, weighted graph, with the sections as vertices. When two sections overlap, the graph represents this as an edge between the corresponding vertices, with edges weighted by the lines of code in the overlap. This representation enables us to apply social network analysis.

After any arbitrary social network analysis, I have a partition of the vertices in the graph, each split into their own respective category. These are the substructures of interest that need to be further inquired into. The network view is designed to permit this.

6.3.1. Concept

I can easily derive a formal definition of the network view from the micro-level view.

Following standard set theory nomenclature [41], a partition P of a set S is a set of non-empty subsets of S such that every element s in S is in exactly one subset. Thus, S is a disjoint union of the subsets. We use the term *cell* to refer to an element of P. Given a section $s \in S$, denote the cell containing s by $[s]$.

Definition (Network View)**.** *Given a micro-level view, let P be a partition of all sections S. Let the cell $p \in P$ be a non-empty subset of S. A network view of this cell is given by the graph of the micro-level view, but every vertex $s \notin p$, along with its edges, is removed.*

Figure 4(b) portrays an exemplary partition with the dashed red lines in a micro-level view. While a partition could be arbitrary, I sketched the figure to represent a partition with maximised shared responsibilities within cells. E.g. sections A, B and C show a strong overlap in (a) compared to A and F. The partition then places the three sections in one cell.

A network view on A, B and C would delete all other vertices, along with their edges, except these three to highlight its structure and erradicate outside noise. On this simplified example, the micro-level view is small enough to offer a clear view on the entirety of its graph as well as

its partition. With growing size, however, the micro-level view easily clutters, necessitating the network view for easier analysis.

Network views showcase a strongly shared theme. An example of a security themed and isolated network view can be seen in Figure 6. It showcases a network view on the security subsystem of Linux. The hierarchical structure of these views and of MAINTAINERS is clearly visible, with `SECURITY SUBSYSTEM` being at the center of the network, connecting to all nodes. Almost all nodes share only one connection, the one to `SECURITY SUBSYSTEM`, but a community-like substructure can be observed among some `KEY`-themed sections, such as `KEYS-ENCRYPTED` and `KEYS-TRUSTED`.

6.3.2. Validity

Next, I explain the various methods to establish the validity of my methodology.

Since I distil a macro-level view from micro-level base data, I need to ascertain that it is performed (a) on correct base data, and that (b) the automated decomposition leads to valid and interpretable results. In the following, I focus on the latter, as the correctness of base data will be discussed in Section 9.2.

The intention of partitioning cells in the network view is to find, as with every clustering approach, sensible groups [47]. In this case, they represent macro-architectural features of the software system. I need to ensure that the resulting decomposition is valid, which comprises internal aspects (influence of clustering technique) and external aspects (semantic validity and interpretability of the resulting cells). Both of these aspects in regard to the validity of the network views will be addressed in the following paragraphs.

Algorithmic Validation Applying a walktrap algorithm which is sensitive to edge weights on the network view is a key step in my methodology. However, the choice of algorithm is subjective. Other clustering algorithms could lead to vastly different, yet valid solutions. Consequently, I need to ensure that the influence of the actual choice is limited.

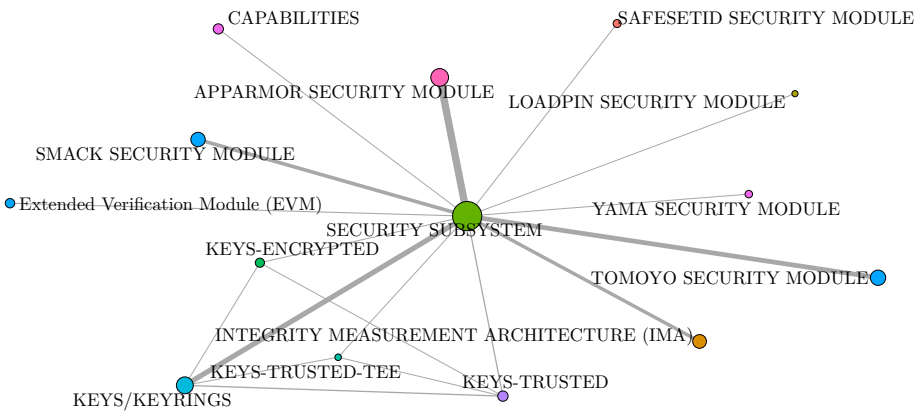


Figure 6: The network view (derived as in Figure 4(b)) for the macro-level view cell "Security Subsystem" from Linux version v5.15.

Table 2: Comparing the walktrap clustering algorithms against the Louvain, Infomap and Fastgreedy algorithm, measuring similarity: P: Purity [102], V: V-Measure [77]. Each value denotes the average value obtained across all analysed versions of the project. For all metrics and projects, $\sigma < 0.049$.

	Louvain		Infomap		Fastgreedy	
Project	P	V	P	V	P	V
Linux	96.0%	83.0%	92.0%	91.8%	96.0%	93.0%
QEMU	96.9%	93.0%	93.9%	96.6%	96.9%	93.1%
U-Boot	98.5%	95.0%	95.5%	94.9%	98.3%	95.0%
Xen	97.5%	97.0%	99.5%	96.4%	97.3%	96.8%

To assess external validity of clustering results by only relying on properties intrinsic to the data, the literature proposes countless measures [102, 77, 81, 35, 95], of which I use the commonly chosen purity [102] (the extent to which two clusterings classify nodes the same way) and V-measure [77] (a combination of homogeneity and completeness of classifications). I compare results obtained with walktrap against results of louvain [17], infomap [78] and fastgreedy [24], as they are commonly employed and readily available in my statistical software of choice [70].

The right side of Table 2 shows the observed similarities between the approaches. Each value is based on computing the average similarity value between walktrap clusters and their counterparts across all versions, averaging over the results for all versions of each subject project.

Since the similarity across algorithms is near-perfect for all combinations, I conclude that the does data contain meaningful structure, and the actual choice of algorithm is therefore inconsequential for the computed results.

Semantic Validation In distilling a macro-level view, I rely on a clustering algorithm. This will, regardless of the algorithmic details, always result in a partitioning of sections into cells. Unfortunately, I cannot rely on any reference decomposition, which makes it impossible to assess result quality using common cluster similarity measures, and thereby to ascertain external validity.

To verify that the computed partition is meaningful from the perspective of developers, I follow a two-stage approach and conduct a quality validation test following the recommended methods of Reyes *et al.* [21]: (1) I used the well-established rewiring procedure [37, 45] on the derived cells, to create randomised versions of the cells that share the essential structural characteristics. All cells containing more than two nodes are retained in the evaluation set, but labels (the section titles) are randomly permuted across cells. (2) I presented *matching pairs* of original and randomised cells to some of the authors of the pre-printed paper that this thesis is based on[4] as expert verifiers (some are intimately acquainted with the subject projects in various roles, and most enjoy senior experience as professional software developers). They performed a binary categorisation of the visual representation of the computed cells, classifying them as either "random" or "real" (each evaluator covered the same graph pairs: 16 for Linux v5.14, 15 for QEMU v6.1.0, 7 for U-Boot 2021-07, 4 for Xen RELEASE-4.15.0). All cases in which the evaluators disagreed (19%), were then jointly discussed *without* knowledge of the actual result, and a community consensus was formed in each case. Finally, the participants compared the

[4]One of the participants selected the example cells and prepared the comparison software package (see the supplementary website for details). Since this resulted in detailed knowledge about the subject cells, s/he did not participate in the assessment, which was performed by four of the other authors of the paper.

evaluation consensus with the actual results, and achieved an agreement in 97.7% of all cases.[5]

Assuming that no interpretable semantics is obtained from the clustering algorithm, it would not be possible for human verifiers to distinguish actual results from randomised versions. Since a distinction was possible with high accuracy, I could ascertain that the employed clustering algorithm yields meaningful structures.

6.4. Macro-Level Views

Computing a partition of the sections, I ultimately derive the macro-level view, as sketched in Figure 4(c) (also shown in Figure 7 based on real data, to be discussed later): Each vertex represents a set of sections (the individual cells of the partition), labelled with the titles of the largest sections (in terms of lines of code) within that cell. In practice, there is always a single largest section.[6] Edges are weighted by the extent of the overlap, measured in lines of code of the involved files.

6.4.1. Concept

Having motivated my methodology, I next provide a formal definition and show the temporal evolution of Linux.

Definition (Macro-Level View)**.** *Given a snapshot of the code repository including a* `MAINTAINERS` *file, let S be the set of sections S. Given a partition P of S, the macro-level view based on partition P is the labelled, undirected graph where each vertex corresponds to one cell in P.*

The vertex corresponding to a cell $p \in P$ is labelled with the set of tuples $\langle title(s), LoC([s]) \rangle$, where s is a section in p such that $LoC(s) = \max\{LoC(s') \mid s' \in [s]\}$. There is an edge between two vertices p_1 and p_2 if there are sections $s_1 \in p_1$, $s_2 \in p_2$ such that s_1 and s_2 overlap. Then, the edge is weighted by the total lines of code of all files within the overlap of s_1 and s_2.

To exemplify that the approach scales to ultra-large software systems like the Linux kernel, and provides intuitively interpretable information, refer to Figure 7: It shows the temporal evolution

[5] Since I presented evaluators with *matching pairs* of random/non-random graphs, the often employed notion of true/false positive/negative results is not applicable, and summaries like precision, recall or related values cannot be inferred.

[6] For sections of equal size, ties can be broken by precedence in lexicographic order.

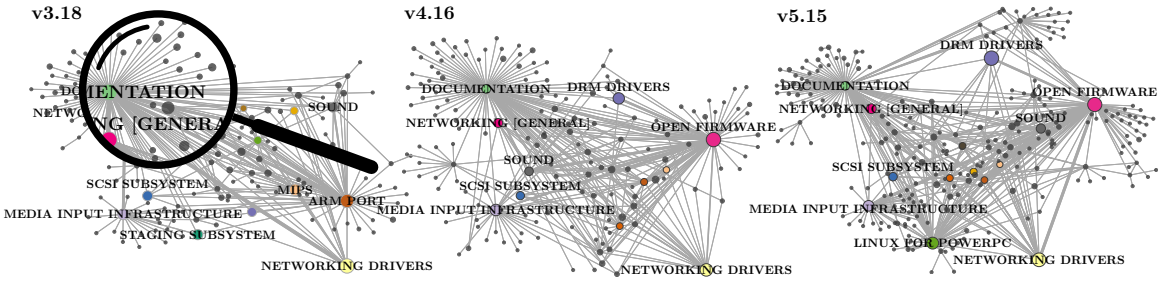


Figure 7: Temporal evolution of macro-level architectural view of Linux kernel, an interpretable macro-level view as in Figure 4(c); node labels correspond to representatives of macro-level view clusters, and coloured consistently throughout, by matching cells over time, from v3.18 (Dec. 2014) to v5.15 (Aug. 2021). Left: A magnifying glass zooms in on the visual details (varying nodes size/edges width). Isolated nodes not shown (<130 per version), to prevent visual clutter.

(over roughly seven years) of the macro-level architecture view for Linux. The macro-level view is able to represent very large systems without overburdening viewers.

6.4.2. Temporal Evolution

An example of three macro-level views can be seen in Figure 7. It portrays the temporal evolution of Linux across three different versions, covering 7 years of evolution. The largest nodes are coloured with the colour staying consistent across releases for easier tracking.

7. Use Case: Conformance

I now present a specific use case where I can employ and extend the introduced methodology. I illustratively align my considerations on the requirements of safety-critical software development, studying the four projects characterised in Table 2; albeit, the overall approach is nonetheless generic.

7.1. Motivation

The technical aspects of software quality can be quantified by countless metrics (maintainability, reliability, issue handling performance…) [100], but they are not the only aspects of quality: Adherence to development processes and architectural specifications is widely accepted in software architecture and engineering as an important contribution to software quality [13], for which it is seen as proxy measure. It is particularly deeply ingrained in safety-critical software development (*i.e.*, software for which failures can lead to catastrophic consequences). Safety standards and norms, *e.g.*, ISO-26262 [46] or IEC-61508 [44] prescribe guidelines to discover and eliminate design and implementation errors as early as possible, which includes suggestions for development and review processes. Establishing trust in processes and architecture requires defined *reference* processes or measurable properties. Relative to these, a *degree* of adherence can then be determined, which allows me to judge if a given project/process satisfies the conditions well enough. This leads to two consequences: (a) A formal definition, or measurable properties, of a process/architecture must be available as reference, and (b) even if a defined reference exists, it can only be consistently enforced when development for a project starts from scratch, and when no external components developed by other means are integrated. Given the importance of re-use and component-based development, these conditions are rarely satisfied, particularly in decentral and self-organised OSS development.

Yet large OSS components such as the Linux kernel, are known for their high software quality [42] as well as consistently enforced processes, and have started to see adoption in use cases such as autonomous driving and medical equipment [51], spacecraft [55], and extra-planetary exploratory vehicles [82]. Similar observations can be made for other low-level base components like QEMU, Xen-Project, and U-Boot. This raises the question of how to objectively ensure (or at least quantify) trust in process and architecture adherence. Based on my automatically derived macro-level view, I propose one possible means in the following. Essentially, I determine to what degree the *defined and inferred* responsibilities for micro- and macro-level clusters match which maintainers show *actual* responsibility for code contributions by merging them into the project. Thus, I define a degree of adherence to process and architecture (in other words: a measure of self-consistency) of a project that is based on time-resolved, self-contained, and without the need of relying on external, subjective assessment.

7.2. Technical Integration Process

In the subject projects, code changes are represented by patches. In those projects, patches are sent to, discussed and reviewed on mailing lists. Before final integration (or rejection), patches may undergo several rounds of revision before they are merged by maintainers. In large projects, author and committer are usually different. What is relevant to my method is which maintainer *initially* picked up a proposed change from mailing list discussions. While I only consider mailing-list based patch discussion and distribution, my approach is independent of this technicality, and only requires (in addition to a responsibility specification) information on the author-committer-relationship, where these two roles are tracked by all current revision control system.

7.3. Notions of Conformance

I introduce two notions of conformance. The first follows directly from MAINTAINERS files. The second is a relaxation, leveraging the macro-level architecture views introduced in this work.

Definition (Micro-Level Conformance)**.** *Given a code repository with a MAINTAINERS file, and a patch integrated by a given maintainer, the integration is* micro-level conform *if the patch targets a file that is in the scope of a section for which the maintainer is responsible.*

Example 3. *For the scenario given in Figure 3, if either Bob or Eve integrate a patch for a file in* `srcs/logging`*, the integration is micro-level conform. This is not the case if they integrate a patch for a file in directory* `scripts` *(other than* `debug.sh`*).*

In practice, I do observe integrations that are not micro-level conform, but close in the following sense: The maintainer M_1 integrating the patch and the maintainer(s) M_2 responsible for the integration as per micro-level conformance are linked by responsibility for *semantically related* sections (*e.g.*, drivers for closely related devices in Linux). When M_1 integrates a patch for which one of M_2 is responsible, this process violation may not constitute the same threat as an integration by a completely unrelated maintainer.

This motivates me to relax the notion of conformance. As in my derivation of the macro-level architecture view, I again assume that the sections in a MAINTAINERS file are partitioned in a meaningful way. For the following definition, it suffices that *some* partition is provided, yet naturally, I will ultimately leverage the partition underlying my concept of macro-level views.

Definition (Macro-Level Conformance)**.** *Given (1) a code repository including a MAINTAINERS file declaring a set of sections S without the catch-all section, (2) a partition P of S, and (3) a patch integrated by a maintainer, as identified by the committer information in the patch. The integration is* macro-level conform *for partition P if one of the following conditions is satisfied:*

- *the integration is micro-level conform, or*
- *the target of the patch is a file in the scope of section s, and there is some section $s' \in [s]$ for which the maintainer is responsible.*

By construction, micro-level conformance implies macro-level conformance. The latter is indeed a strict relaxation of the former. Let me point out that micro- and macro-level conformance combine aspects of adherence to a development *process*—ultimately, entries in MAINTAINERS files specify who to send a patch to—*and* a software *architecture*—the derived macro-level view is an abstract architecture view, and macro-level conformance implies adherence to an architecture. Therefore, my proposed measures provide a combined view that unites aspects of process and architecture.

7.4. Analysis Pipeline

The upper part of Figure 1 shows the analysis pipeline for my use case. Based on the source tree and the developer mailing lists as input artefacts, I employ the third-party tool PaStA for artefact parsing. Specifically, my analysis leverages historic archives of 270 publicly available mailing lists for my subject projects, where mailing lists are primary communication resources [53, 49]. I integrate this data with the macro-level view, distilled by my methodology, and then analyse whether patches have been conformingly integrated.

7.5. Verification: Maintainer Survey

To complement the semantic validation of the derived macro-level architecture views, I conducted a survey among maintainers of the Linux kernel. Based on my own knowledge of the project, I selected nine senior maintainers from different areas of responsibility, six of which responded (a response rate of 67%).[7] I deliberately focused on a narrow, but judicious selection of maintainers, and avoided a wider distribution of the survey that would refer to personal data on general mailing lists (a) out of research-ethical considerations, and (b) to respect the wishes of the Linux community which "*welcomes developers who wish to help and enhance Linux*", but "*does not appreciate being experimented on*".[8] The exact survey formulation is documented on the supplementary website, together with the verbatim responses. Since the statements share considerable overlap, I present here a selection that covers the common tone. Prior to distributing the survey eMail that references two maintainers by name, I obtained their consent.

7.5.1. Survey Design

To avoid introducing interpretation bias, I did not disclose any technical details of how I distil the macro-level architectural view, but only asked to judge correctness of the results. To not place undue (from their point-of-view: unproductive) load on the maintainers, I restricted the questions to two carefully selected scenarios that represent important cases. Since the validity of the macro-level decomposition has already been well established in Section 6.3.2, I designed the survey so that I could infer information relevant for the use case discussed in this chapter.

I selected two commits that were both *not* integrated by the directly responsible person as specified in MAINTAINERS. In one case (referred to as case A), the integration was performed by another maintainer from *within* the macro-level view cell identified by my approach, the other (case B) by a maintainer *outside* the macro-level view. This difference was not communicated to the respondents. Assuming that my architectural view is accurate, I expected that the integration in case A would be seen as correct, whereas case B would be identified as incorrect, or that an explanation why an exception is justified would be provided. The survey provided sufficient detail for interviewees to easily reference all associated commits, related discussions on the project-specific channels, and the relevant portions of the MAINTAINERS specification.

7.5.2. Responses for Case A – macro-level view conforming

Commit 4499d488f violates the specification in MAINTAINERS, but matches the cell in the architecture macro-level view (the conformity time series of the corresponding feature is illustrated in the bottom right part of Figure 8). Therefore, it represents a case where having additional context information provided by my methodology is preferable to strict micro-level conformance.

The patch committer, Jani Nikula, pointed out that the portion of Linux kernel addressed by the patch is group-maintained, which means that "*the maintainers listed in MAINTAINERS oversee development, send pull requests [...]*", and there are "*dozens of people with commit access[...], with documented merge criteria*". The specific, albeit slightly outdated documentation is available online.

[7] I presented an earlier version of my results at the Linux Conference Australia for Linux, which triggered interest of maintainers that requested information to learn about how "their" subsystem compared to other subsystems in terms of conformance.Additionally, the average response time of maintainers was slightly below two hours, which I take as an indication of considerable community interest in the topic.

[8] This was communicated unmistakably in a statement issued by Linux Foundation fellow Greg Kroah-Hartman, one of the most senior kernel maintainers, in response to academic actions perceived as inappropriate by the kernel community.

The group-maintenance structure was pointed out by all respondents, for instance Jonathan Corbet ("*This one is easy; DRM is group-maintained, and they are all empowered to accept patches throughout the subsystem.*") or Lee Jones ("*DRM is a tricky one as it's group maintained.*"), albeit they varied in their assessment on how hard it is even for kernel maintainers to recognise the circumstances.

The respondents agreed that the integration was performed correctly, despite formally violating the verbatim MAINTAINERS specification. We take this as confirmation of the utility of my macro-level view that was compatible with the (implicit) group maintenance structure inferred from the micro-level view data.

7.5.3. Responses for Case B – invalid integration

Commit 5dc33592e was integrated by a maintainer *outside* the macro-level view cell, thus violating the maintainer specification provided by the project *and* my architecture decomposition. The patch author, Tetsuo Handa[9], pointed us to a discussion of the change, and explained that he sent the irregular pull request because the responsible maintainer, who "*did not like the patch*", was not responsive. Since the patch fixes a pressing issue, the pull request submitted to the top-level maintainer, Linus Torvalds, included a detailed explanation of the situation, and was eventually merged out-of-band. We take this as confirmation that the derived macro-level view was violated by the integration, and that this violation was correctly detected by my analysis. Yet this is it based on a rationale that could not be reconstructed without human involvement.

This perception was shared by others: Jan Kiszka remarked that "*[..] there can be sideband agreement between maintainers[...], specifically when a patch touches multiple subsystems*", adding that this "*[...] is documented in the pull request message sent to Linus or another maintainer*". Paolo Bonzini confirmed out-of-band integrations are justified when necessary; in his view, the commit follows a "*[...] relatively common way to handle patches that for some reason were dropped by the directly relevant maintainers*". Jonathan Corbet underlined that "*[...] it has always been possible for developers to touch any part of the kernel tree when justified*", albeit in this case that he deems "*a bit questionable*", he would have preferred to "*[...] find a way to convince the lockdep maintainers to take it*".

Overall, all respondents agreed that the integration was out-of-band, but the exception was properly justified. We conclude that the violation of the derived macro-level view can *not* be attributed to a weakness of the method, but instead highlights the need for some flexibility in validating decomposition results.

7.6. Conformance Observations & Evolution: Linux Overall and Linux Features

The fraction of micro-level and macro-level conform integrations (relative to all integrated changes) is shown as a time series for Linux (and some feature-resolved plots that I address later) in Figure 8.

While the measurements are affected by varying amounts of noise that cause local variations at the scale of weeks, any changes in the longer-term *trend* curve—illustrated by the solid lines computed using penalised regression splines in a generalised additive model—agree well with the time scales of years that evaluators would typically be interested in for judging project maturity.

[9] T. Handa maintains section "Tomoyo Security" (visible as a node in the network view in Figure 6). The commit touches section "Locking Primitives", *not* present in the "Security Subsystem" cell and therefore considered an invalid integration.

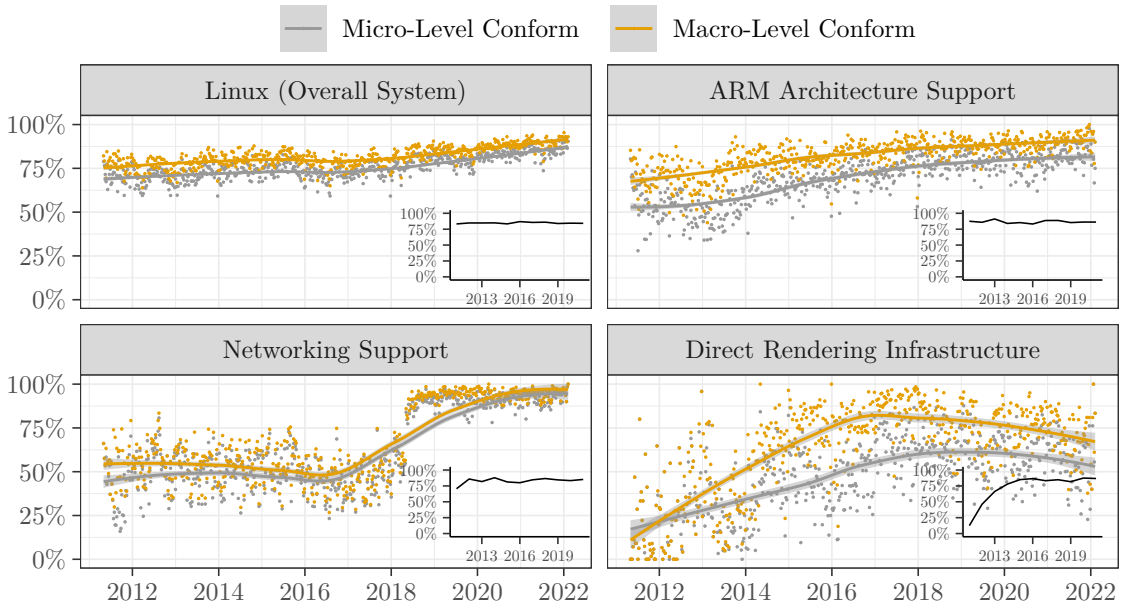


Figure 8: Process-conform integration over time for the overall Linux kernel mailing list (top left), and selected components/features of the kernel (as represented by a feature-specific mailing list). Dots represent observed measured values as provided by my method, solid lines a smoothed trend. The insets show how similar the time series are, resolved over time (see the main text for a detailed explanation).

Notably, micro-level and macro-level measures describe highly similar trends, which is partly evident from visual inspection of Figure 8. To quantify the similarity of the micro-level and macro-level time series in Figure 8, I use the normalised compression distance [22]. This measure enjoys a sound theoretical foundation, based on Kolmogorov complexity, but can be readily computed numerically [61]. An excellent agreement of typically more than 85% (100% implies point-wise identical curves) in yearly granularity is shown in the insets of Figure 8, and ascertains that both views agree in their trends.

Since the macro-level view is based on substantially fewer artefacts (tens to hundreds of cells) than the micro-level view (hundreds to thousands of sections), the goal of *faithful abstraction* is satisfied by the macro-level. The same observations hold for all other subjects projects seen in Figure 9.

The goal of this work is to provide a quantitative, objective basis to assess properties and evolution of conform code integration, but *not* to perform a specific judgement of a project, since this requires criteria defined for the given context (*e.g.*, a safety architecture or development process certification).

Nonetheless, let me sketch one idea of how the available information could be leveraged: All subject projects increase in code volume and number of contributors over time, and Figure 7 shows a clear growth in complexity of the Linux project over time in this respect. Yet I can also observe that conformance *improves* in the same time range, which indicates that the involved processes have not yet reached their limits, or that the capabilities of maintainers have improved over time. In particular, Linux and Xen show a large fraction of conforming integrations, which matches their reputation of projects with a high degree of process maturity.

Linux. The top left plot in Figure 8 shows the overall Linux system. It consists of the conform integration ratio on *all* Linux lists. It shows a clear upward trend, with the macro-level conformity consistently leading as a notable improvement in conformity. This shows us that integrations at macro-level conformity play a significant role in the general conformity of the project.

While other observations on the generic behaviour of the subject project could be made, let us assume a different point-of-view. Software is often structured in the form of product lines [8], where low-level components in general, and Linux in particular, expose substantial feature variability [80]. Therefore, it is appropriate to apply my method to assess qualities of the implementation of specific features. Safety-critical systems, for instance, are typically built around embedded computing components, which in turn usually employ a strongly tailored configuration that only activates necessary features. Consequently, their quality properties are of interest, whereas properties of unused features can be neglected since they do not impact the resulting system. The top right and the bottom graphs in Figure 8 show the evolution of conform integration for three exemplary Linux features. Clearly, observe different trends for specific features are observable that do not match the observations of the overall project, and require consideration and interpretation.

ARM Architecture Support. The trend for ARM architecture support (top right in Figure 8) is different than for the overall kernel. For one, the conformity has improved at a higher rate compared to the overall system; additionally, the difference between micro-level and macro-level view conform integration ratios is more pronounced than for Linux in general. Especially during the first years captured in the measurement, this may lead to a more negative assessment of (local) quality than is actually merited. The overall increase in macro-level conformity, on the other hand, is less pronounced, and reaches levels comparable to the overall system from about 2014 onward. The ARM architecture needs to address many closely intertwined technical aspects [59], *e.g.*, following from vendor-specific implementations of the same generic standard. This creates many opportunities for merging related patches among domain experts, a scenario that is better reflected by the macro-level architecture view.

Networking Support. For network devices (bottom left in Figure 8), we observe a very close relationship between micro-level and macro-level conformance. Until 2018, conformance in both views is inferior compared to the overall system, but only to rise to superior conformity afterwards, with a near-perfect ratio. The close agreement of both measures suggests that the marked change in trend is not caused by an issue related to collaboration. Manual inspection of the change history shows that a maintainer addition is documented for April 2018, when David S. Miller was added to the `NETWORKING DRIVERS` section in the MAINTAINERS file. As one of the most senior Linux network engineers (and one of the earliest contributors to the project), the change corrects a factual omission in the responsibility specification, and adapts the specification to actual reality. Such context knowledge is, obviously, crucial to obtain an informed conformity appraisal.

Direct Rendering Infrastructure. Finally, consider the behaviour of "Direct Rendering Infrastructure" (bottom right in Figure 8), a feature that enables application access to 3D accelerator hardware. Not only are there very pronounced differences between the macro-level and micro-level conformity trend, but the code also starts out with a seemingly catastrophic conformance that never catches up with the overall system. This is explainable, as one of the feature developers communicated to us: The feature is "*[...] maintained using a committer/maintainer model*

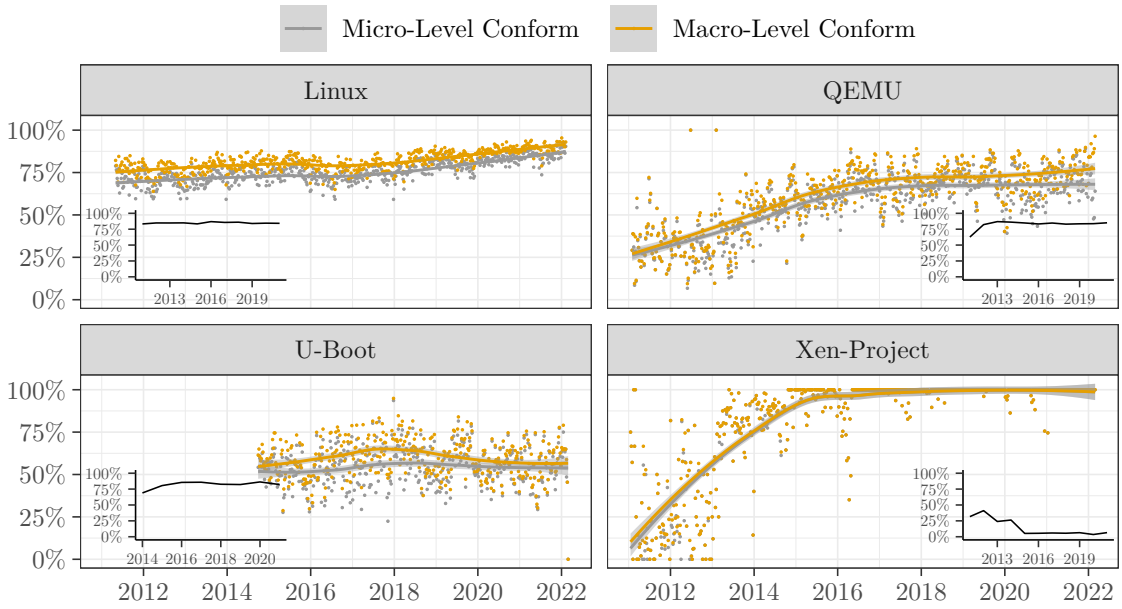


Figure 9: Process-conform integration over time for all projects. Dots represent observed measured values as provided by my method, solid lines a smoothed trend. The insets show how similar the time series are, resolved over time (see the main text for a detailed explanation).

with shared tooling. There are literally dozens of people with commit access to each, with documented merge criteria. The maintainers listed in MAINTAINERS oversee development [...]". While responsible persons are listed in the file, the actual responsibility is *shared*, thus favouring macro-level conformity over micro-level conformity.

7.7. Project Conformance

Not only did I analyse Linux and its features, I also researched the general conformity behaviour of QEMU, U-Boot and Xen. Since these projects do not compare to Linux in size, I conducted a general conformity analysis, not on feature level. The results can be seen in Figure 9. Linux as an overall system is listed again top left for comparison.

QEMU. For QEMU, we observe a relatively low ratio at the beginning of the analysis, rising to almost Linux like conformance levels at the end. Noteworthy is the strong relationship between micro-level and macro-level conformance, with macro-level improving even more over micro-level over time.

U-Boot. U-Boot has the worst overall micro-level and macro-level conformance ratio, stagnating around 50% since the beginning of my analyses. U-Boot is the second smallest of all analysed projects in this work. Perfect adherence to the development process does not seem to be a primary concern in this particular case. Nevertheless is the project known for its high quality of code, so despite the fact that the de-facto implemented development process does not fit MAINTAINERS situation, it still works out well in this case.

Xen. Xen has an exceptionally high micro-level conformance. With the macro-level conformance being a relaxation of micro-level conformance, it does not have room for notably more conformance improvement, so both measures maintain a consistently high conformance. Part of the reason for this exceptionally high conformance ratio could be due to the small size of Xen, being the smallest of the four analysed projects with significantly less integrations than the others.

8. Reproduction

Science strives to gain new insights and publish them to further human knowledge and broaden its horizon. To battle forgery, sloppy research practice or faulty test results, all publications should be reproducible to enable reproduction experiments and confirm their validity. The Association for Computing Machinery (ACM) states that "an experimental result is not fully established unless it can be independently reproduced" [10]. It considers an experiment fully *reproducible* when a different research team can achieve the same results with the same experimental setup. If, additionally to the different team, a different experimental setup also wields the same results, the experiment is considered *replicable*.

The precise definition given by the ACM is as follows:

Definition (Replicability)**.** *"The measurement can be obtained with stated precision by a different team, a different measuring system, in a different location on multiple trials. For computational experiments, this means that an independent group can obtain the same result using artifacts which they develop completely independently."*

Reproducible and replicable experiments are without a doubt extremely important for science. They are, however, often neglected. A study from 2016 shows that around 70% of scientists have failed to replicate work from other scientists and more than 50% had problems reproducing their own results [12].

To ensure the replicability of the results from this work, I will deliver a fully automated reproduction package. This section will guide through its composition, components and usage.

8.1. Docker

Container-based virtualisiation is a virtualisation approach that offers virtualisation at operating system level. It uses the host kernel to run multiple virtual environments, often referred to as *containers* [20]. By virtualising on operating system level, only one operating system kernel needs to run and can manage the containers. It does not need to have multiple kernels employed redundantly. The containers run like normal processes on the host machine, managed by the main operating system, in its own isolated environment.

Docker containers are built through *Docker images*. A docker image stacks multiple data layers onto a base image to create a specialised environment. E.g., if the user uses a Ubuntu base image and clones a git repo, an additional data layer containing the repo is added to the image.

An own docker image can be created through *Dockerfiles*, which provide a simple script, similar to a Makefile, that defines how the image is built [18]. I will make use of this "recipe-like" description of a script build my experimental setup onto my base image and then run the experiment to reproduce the results.

8.2. Reproduction Package

Since the purpose of the reproduction package is to reproduce the results from scratch, I need to reproduce the data from scratch as well. These reproduction pipelines does this by first cloning the PaStA repository and initialising the projects Linux, U-Boot, QEMU and Xen as submodules and updating them to the most recent release, to get the necessary base infrastructure.

The projects are then analysed to produce the project characteristics. This step prepares the mailing list data and then analyses them in various aspects, including conformance and recreates all graphs, including the randomised ones for the GUI's from my validation process from 9.

All this can be achieved by simply running the `docker/build.sh` script in the PaStA repository. The docker image uses the user `pasta`. All results can be found in a directory called `results` in this user's home directory from this docker image.

9. Discussion

9.1. Threats to Validity: Internal Validity

Parsing. PaStA provides a custom parser for MAINTAINERS specifications across subject projects. However, for all subject projects, the script `get_maintainer.pl` that parses the MAINTAINERS file, and the content of the file itself, have changed over time. Both may contain project-specific adaptations, and use project-specific conventions. Since I do not simulate the exact semantics for each project and point in time, this could lead to mis-parsing entries.

To boost performance, I employ the MAINTAINERS parser by PaStA, rather than the project-provided parsers. To ensure correctness, I carefully compared parser outputs on a randomly sampled subset of 600-5,000 patches (depending on the project). This produced identical results in about 90% for all projects, rendering this a minor threat.

Commits. In my use case analysis, I rely on the tool PaStA, and therefore only consider commits that were discussed on mailing lists prior to integration. This allows for quantifying and comparing mailing list-specific conform integration ratios (cf. the supplementary website), but also misses commits that cannot be mapped to an artefact on a list. Nevertheless, the vast majority of commits in repositories can indeed be mapped: For U-Boot, QEMU, Linux and Xen, I quantified the commit coverage[10] to 96.7%, 94.6%, 90% and 78% respectively. Hence, able to to group results by mailing lists (as in Figure 8) outweighs incomplete commit coverage.

9.2. Threats to Validity: External Validity

Stale MAINTAINERS. Large open-source projects with a self-organised community and active development contributions need to establish processes to split the code base and delegate work. The MAINTAINERS-approach has established itself as good practice for submission guidelines, proven by its long-term usage for Linux and its adaptation by numerous other projects.

To keep development in large projects running fluently and minimise organisational arrangements between maintainers, it is within the community's best interest to encourage usage of and ensure maintenance for the MAINTAINERS file. Patches are supposed to reach the right people actively acting in the role as maintainer or developer for the affected sections.

As such, the structure of MAINTAINERS is kept simple and easy to maintain and regularly updated for all analysed projects. There is a strong incentive for developers to keep MAINTAINERS up-to-date: I find that Linux provide updates around 12 times per week, QEMU and U-Boot around 2-3 times, and Xen, as the smallest, roughly every third week. This differentiates the file from other project documentation artefacts, which often suffer from low maintenance and outdated descriptions.

My methodology relies on areas of declared responsibility, as exemplified by MAINTAINERS files. One threat to validity is that these files may be stale. Yet I deem this threat minor, as this claim is easy to refute by determining the update frequency of the MAINTAINERS file.

I am therefore confident that my approach to use MAINTAINERS as a ground-truth to depict an accurate depiction of the project's structures is grounded in sound assumptions.

Generalisability. A further concern is the generalisability of my methodology beyond MAINTAINERS. I see no reason why the approach could not be extended to related artefacts, such as the Google `OWNERS` or GitHub `CODEOWNERS` files (discussed in related work). I am confident that my methodology can be generalised to projects with other means for specifying areas of

[10] Ratio of the number of commits that can be mapped to mailing list artefacts vs. number of total commits.

responsibility. While I rely on mailing list analysis, the pipeline could easily be adapted to other means of identifying relevant patches.

I am therefore confident that my approach to use MAINTAINERS as a ground-truth to depict an accurate depiction of the project's structures is grounded in sound assumptions.

9.3. Threats to Validity: Data/Construct Validity

Impartial participants. In semantic validation (Section 6.3.2), a common threat is an insufficient impartiality of the participants. To counter this threat, I designed the rewriting experiment such that the participants got to choose between a partition produced by the walktrap algorithm, against a randomised partition. Thus, the experiment was set up such that the participants could not anticipate an answer that complies with a gold standard, such as the question which macro-level view seems more meaningful. As a consequence to this design, I cannot provide metrics like precision or recall.

In validating my use case (Section 7.5), one threat is that the number of open-source developers is too small, or the focus on a single project too narrow. Yet as laid out, I am constrained by the codes of conduct, as unsolicited surveys among developers are neither ethical nor appreciated.

Projects studied. I base this analysis on four projects, which raises the threat of generalisability. Yet as long as new projects provide a parseable declaration of areas of responsibilities, I could adapt my methodology accordingly. Via my detailed mixed-method verification, I further weaken the threat of a small sample size.

10. Conclusion

Software reverse engineering and component detection has been a researched problem for more than 20 years, but still requires heavily time consuming work and attention from developers. It has long been suggested that ownership information provides extremely valuable help when trying to understand a system, since it contains implicit domain information. Furthermore, the practice has proven its benefit for maintenance and development processes and continues to be implemented in more and more projects.

In my multi-stepped approach, I presented a method to turn socio-technical information from ownership artefacts into semantically meaningful and stable micro-level and macro-level architecture views. Especially, the method is independent from the implementation languages, and therefore widely applicable.

By discussing concrete use cases, I have shown that it can assist in solving ongoing and long-standing challenge in the industrial deployment of OSS. Finally, using a thorough mixed-method validation, I systematically assessed my results.

Future work will focus on improving and refining the method, such as extending it to formats beyond MAINTAINERS, and to design concrete tools can provide utility to OSS projects.

Acknowledgements. I want to thank Ralf Ramsauer, Stefanie Scherzinger and Wolfgang Mauerer for the extensive support and fruitful discussions. I furthermore want to thank Thomas Kirz for his vast support with TIKZ-Visualisations.

A. Appendix

Here we present a few selected network views from all four projects.

A.1. Linux

These are network views from Linux v5.15.

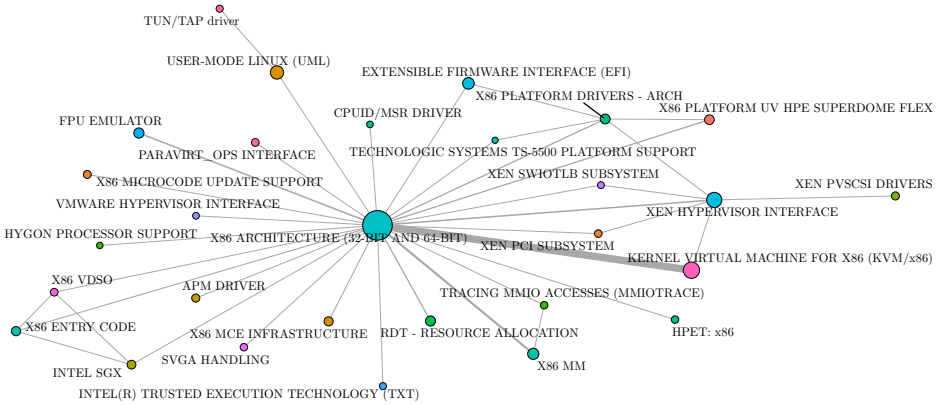


Figure 10: A network view for X86 and Xen themed sections.

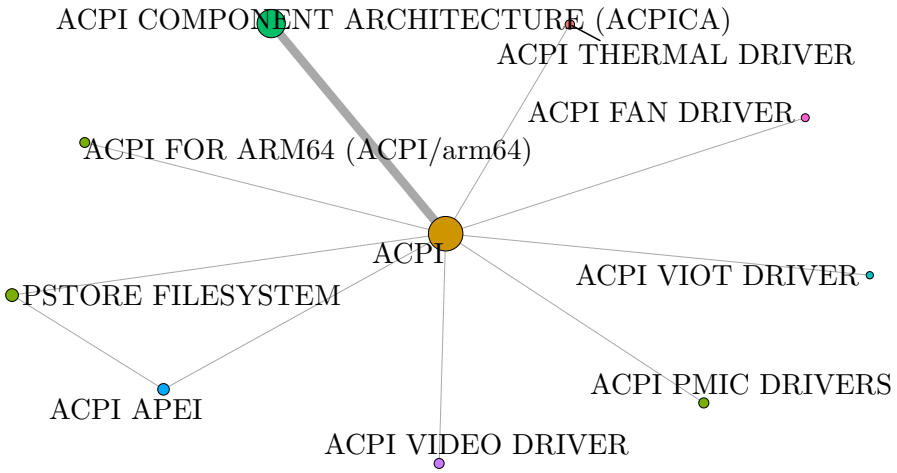


Figure 11: An ACPI themed network view.

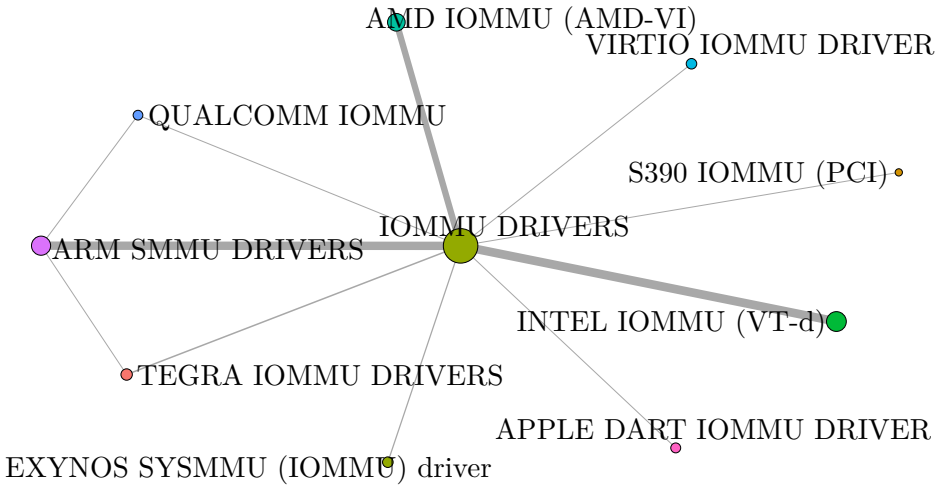


Figure 12: A IOMMU themed network view.

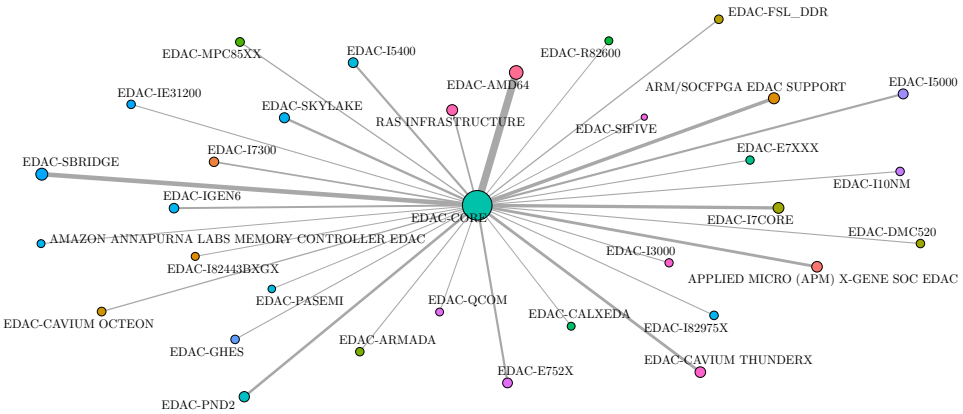


Figure 13: An EDAC themed network view.

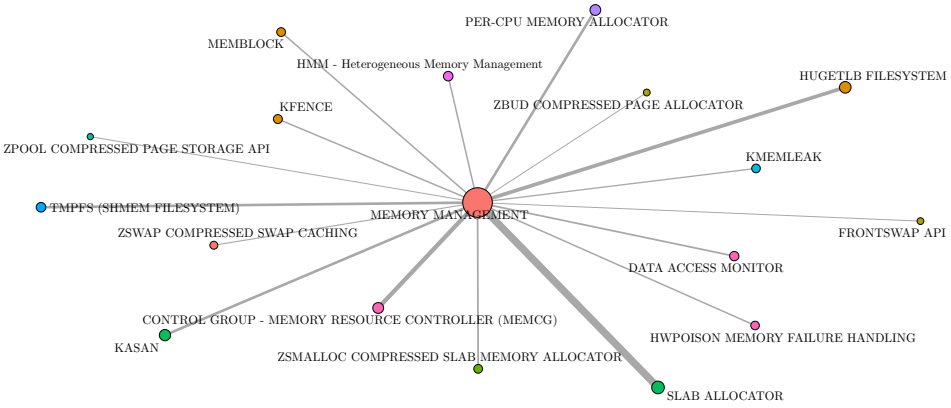


Figure 14: A network view for general memory management.

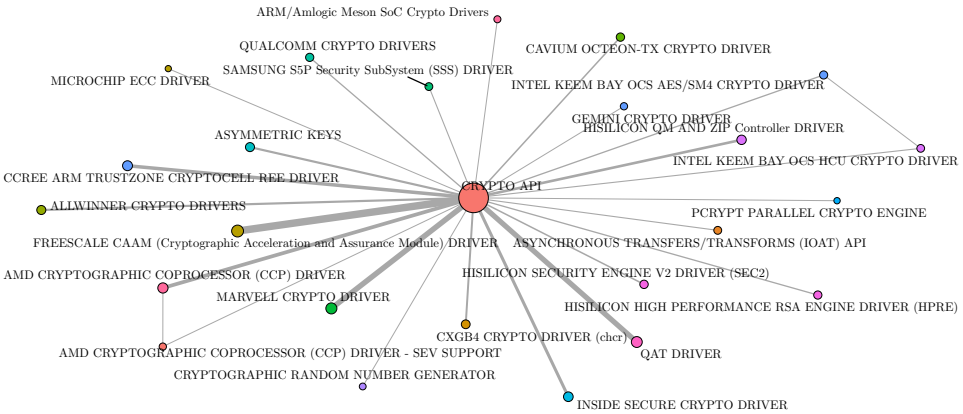


Figure 15: The network view for cryptographic sections.

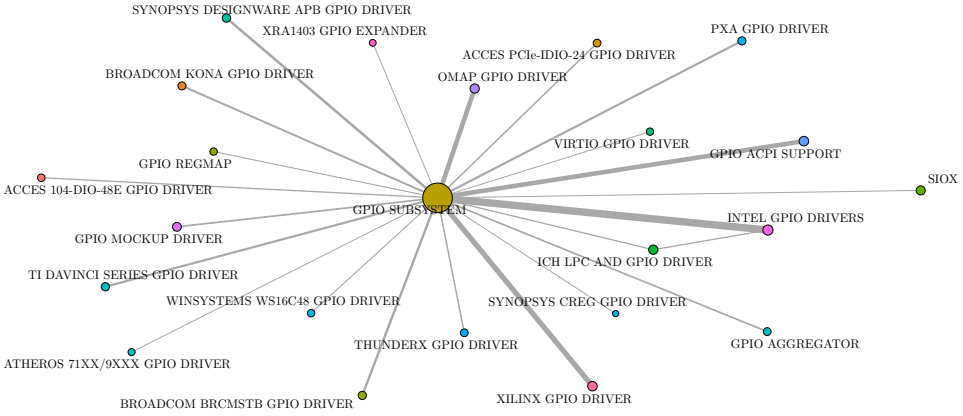


Figure 16: A GPIO themed network view.

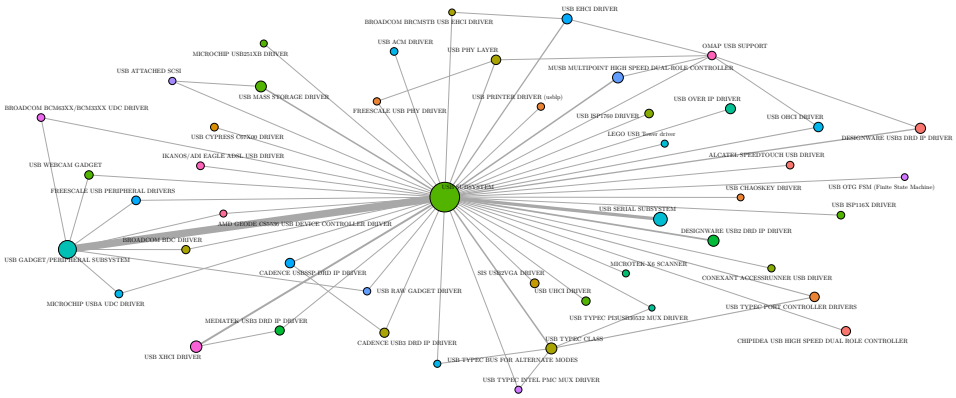


Figure 17: A network view for the USB subsystem.

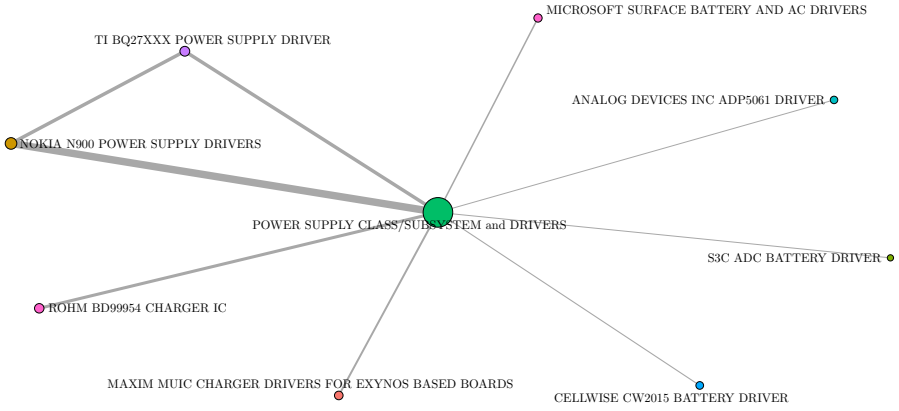


Figure 18: A network view for power supply sections.

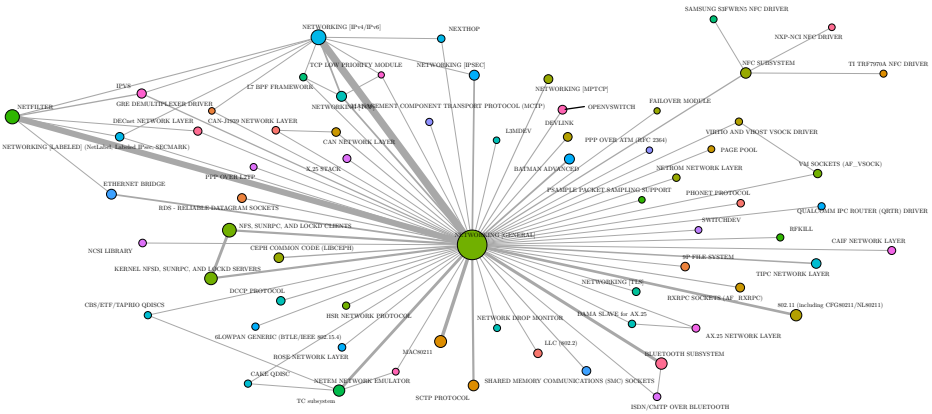


Figure 19: The general networking network view. This is a very large and central cluster for LINUX.

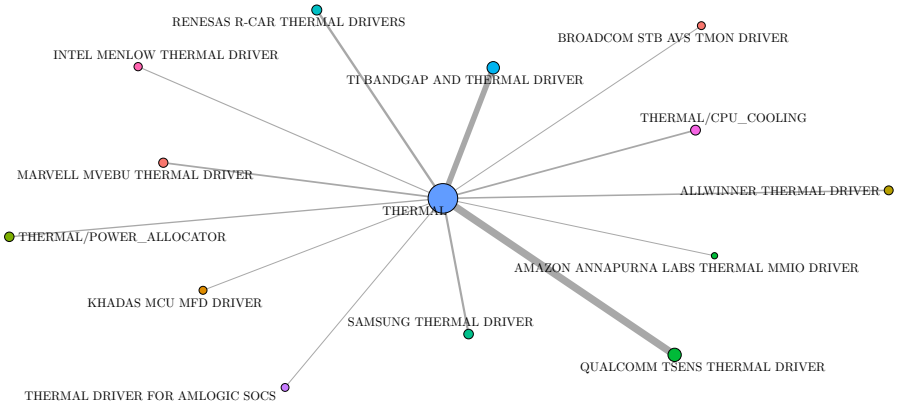


Figure 20: A thermal themed network view.

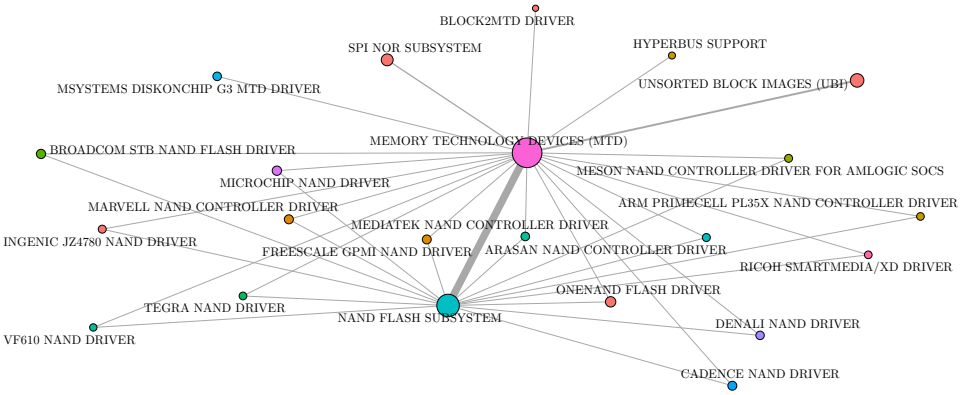


Figure 21: A memory technology and nand themed network view.

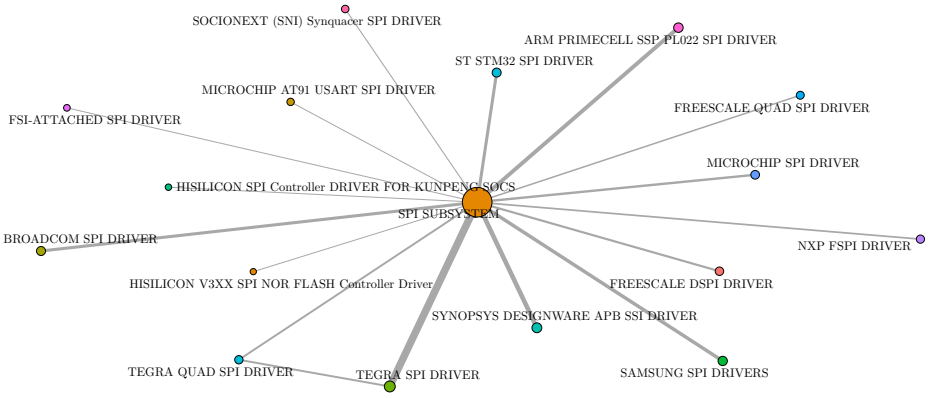


Figure 22: A SPI themed network view.

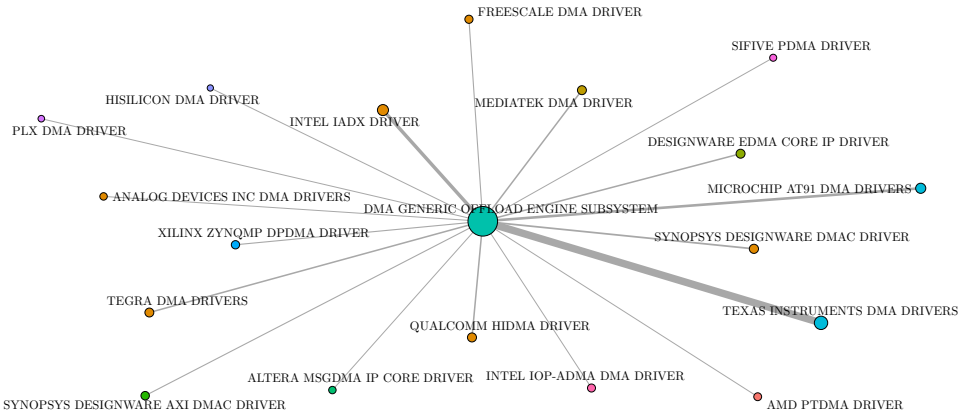


Figure 23: A DMA themed network view.

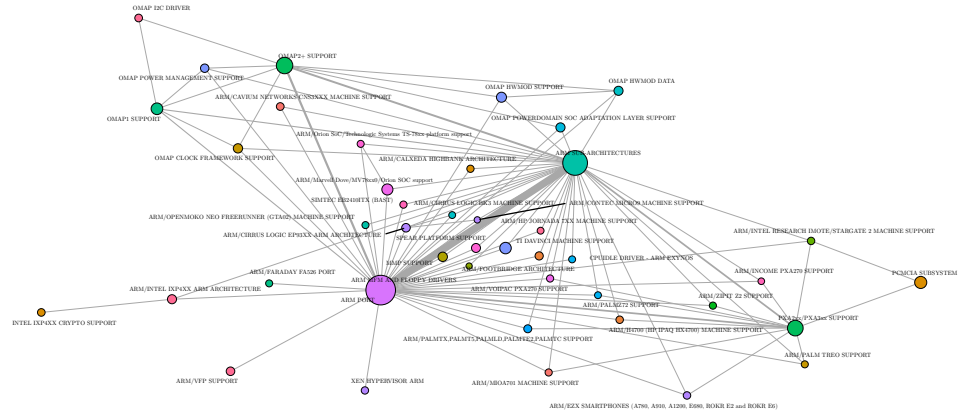


Figure 24: An ARM themed network view.

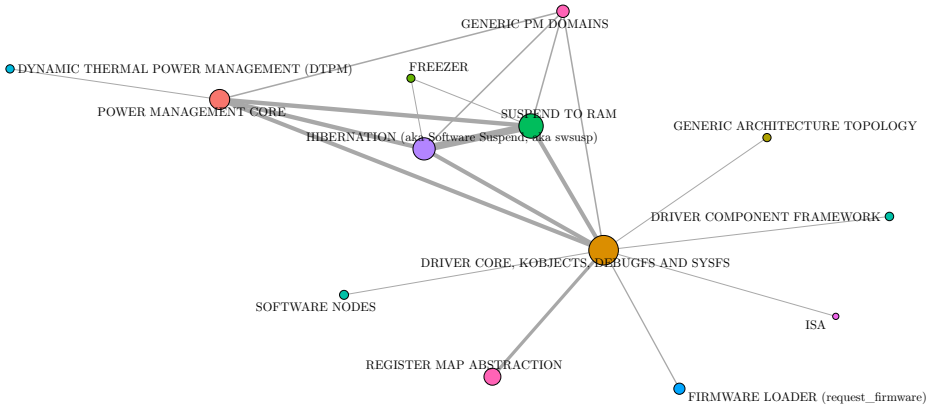


Figure 25: A network view for power management.

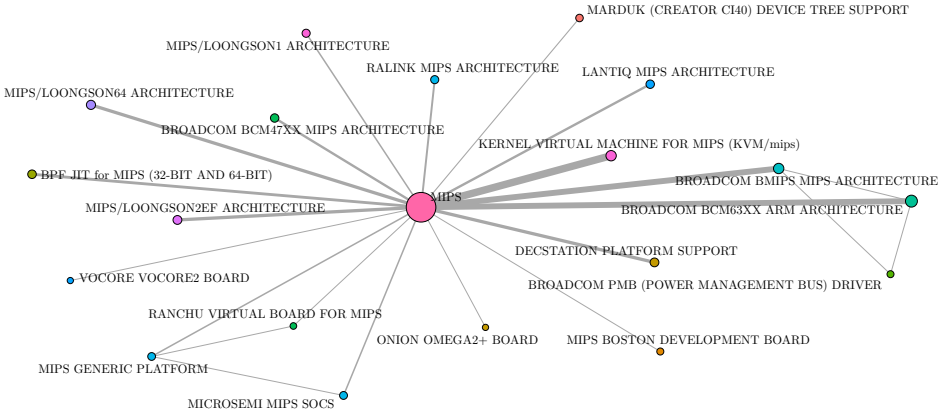


Figure 26: A MIPS themed network view.

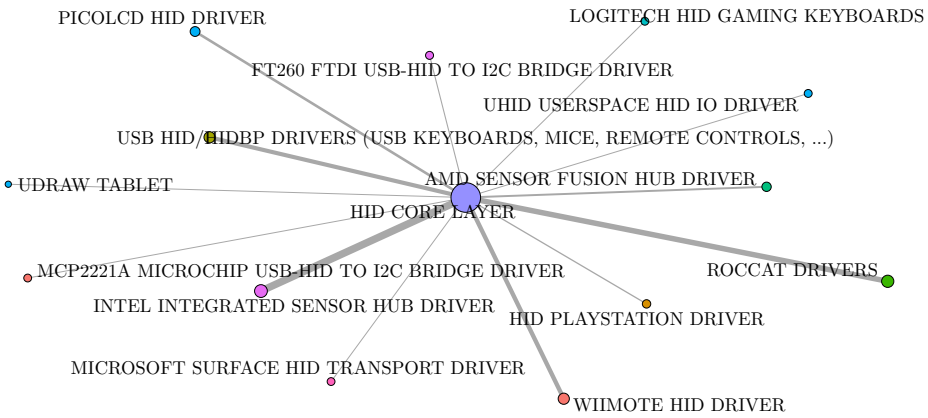


Figure 27: A HID themed network view.

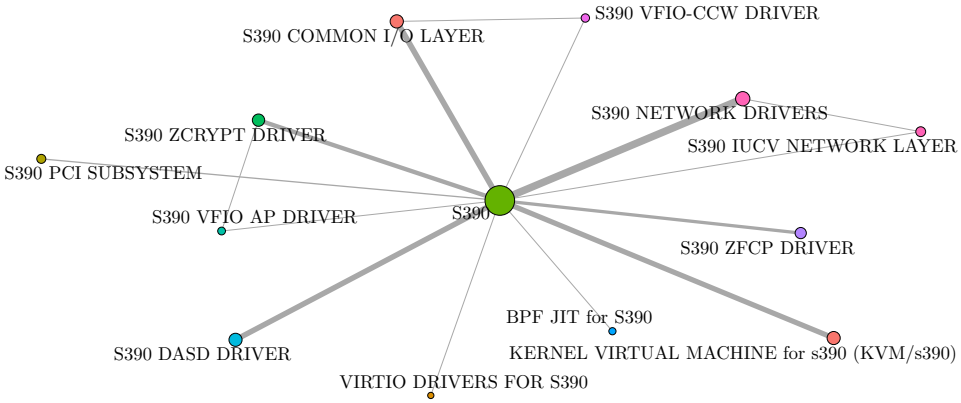


Figure 28: A S360 themed network view.

A.2. QEMU

Here we present a few selected network views for QEMU 6.1.0.

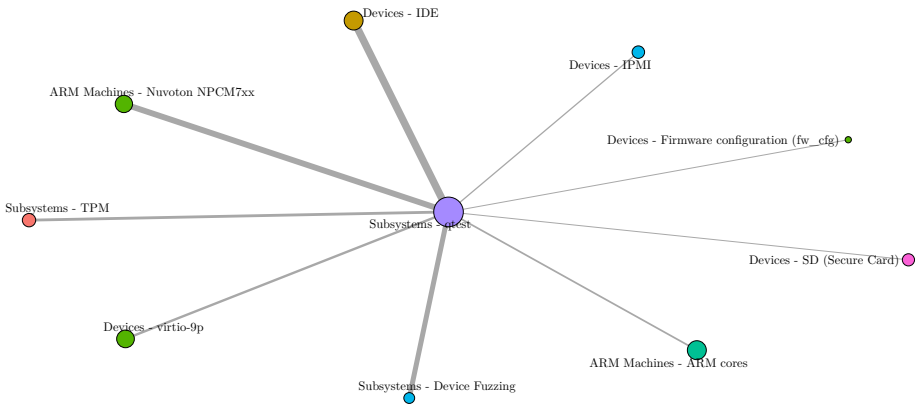


Figure 29: A network view with the subsystems for qtest.

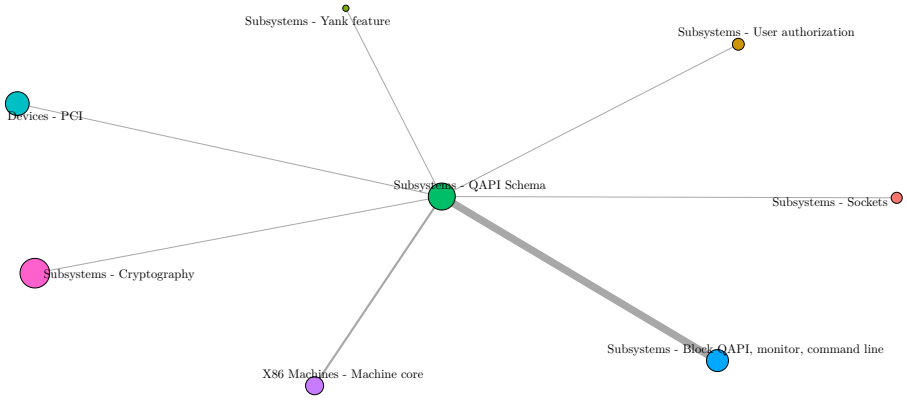


Figure 30: A network view for QAPI themed sections.

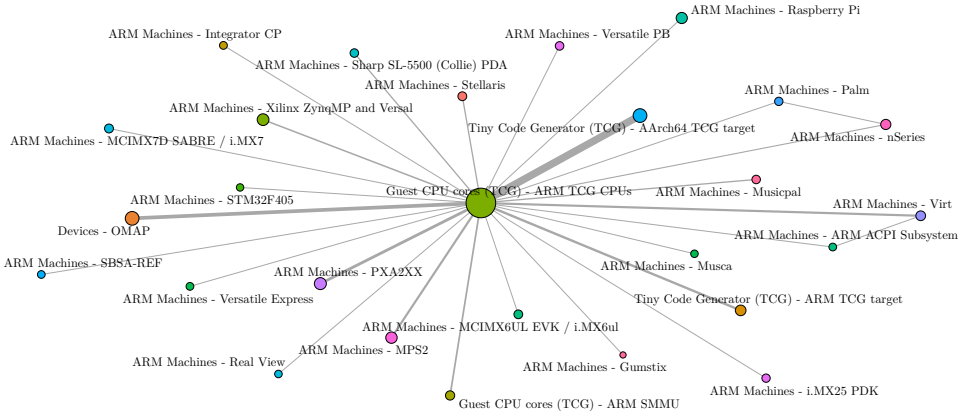


Figure 31: A network view for guest CPU's.

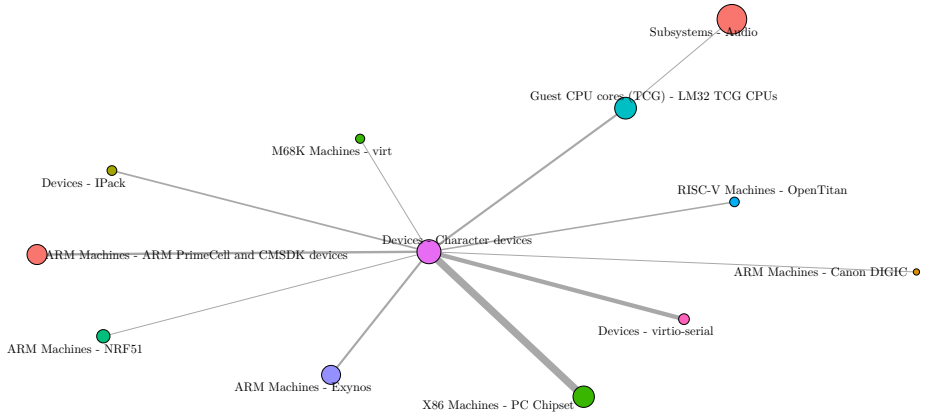


Figure 32: A network view for character devices.

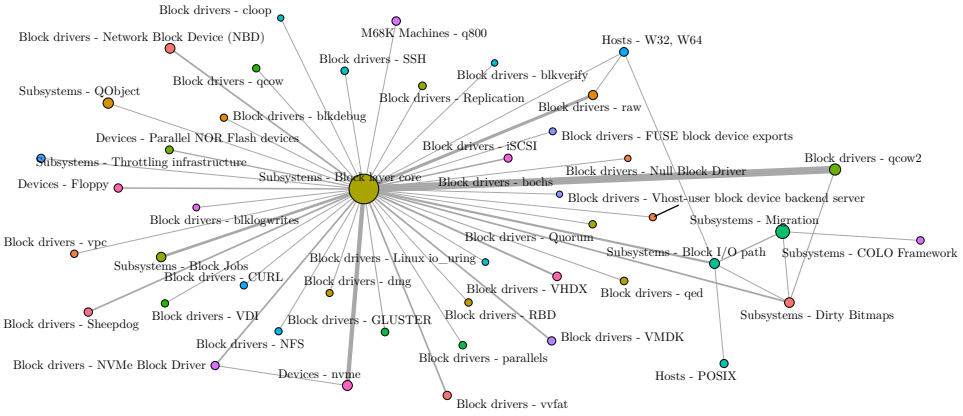


Figure 33: A network view for block drivers.

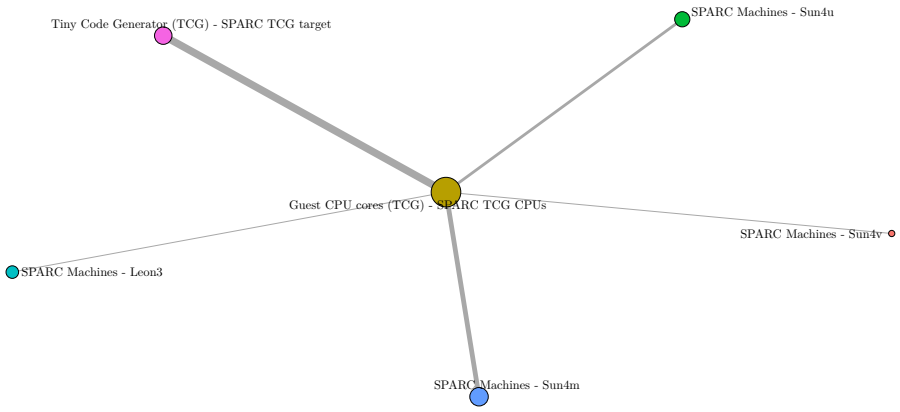


Figure 34: A SPARC themed network view.

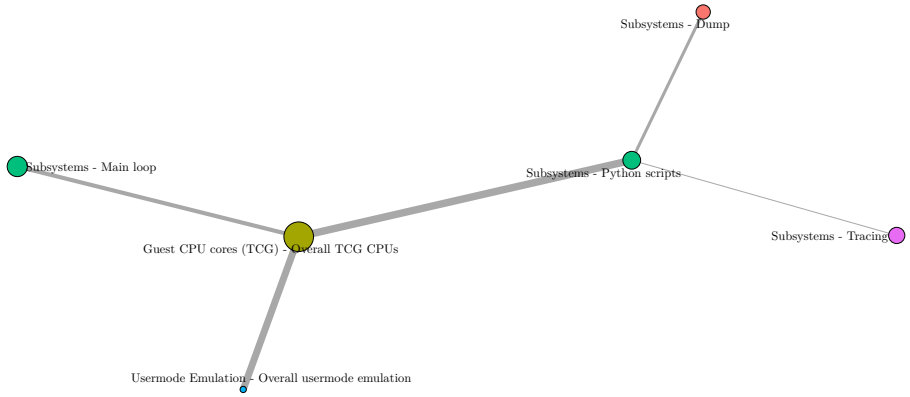


Figure 35: A network view for general TCG guest CPU's.

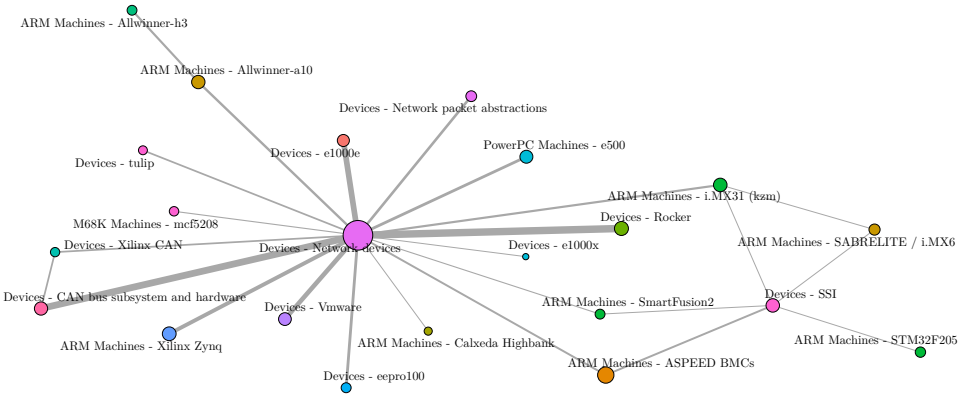


Figure 36: A network view for network devices.

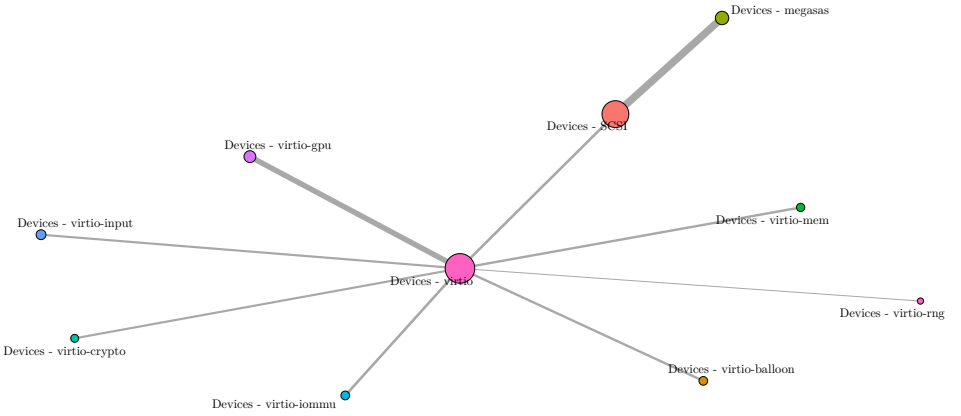


Figure 37: A network view for virtio devices.

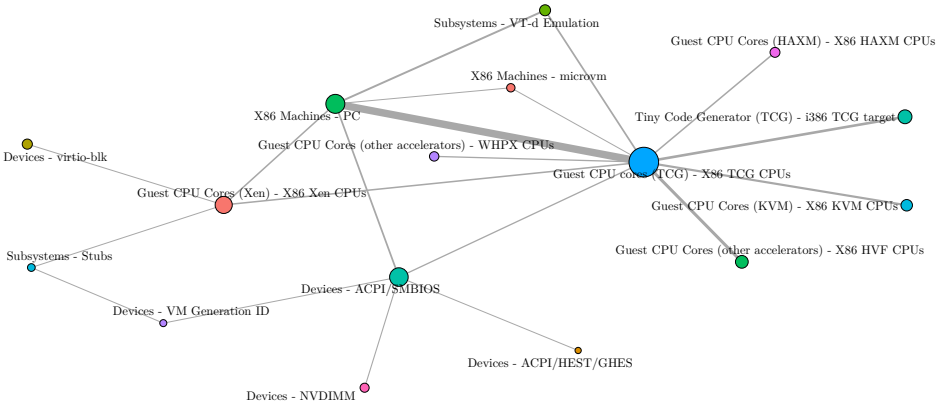


Figure 38: A network view for X86 specific sections.

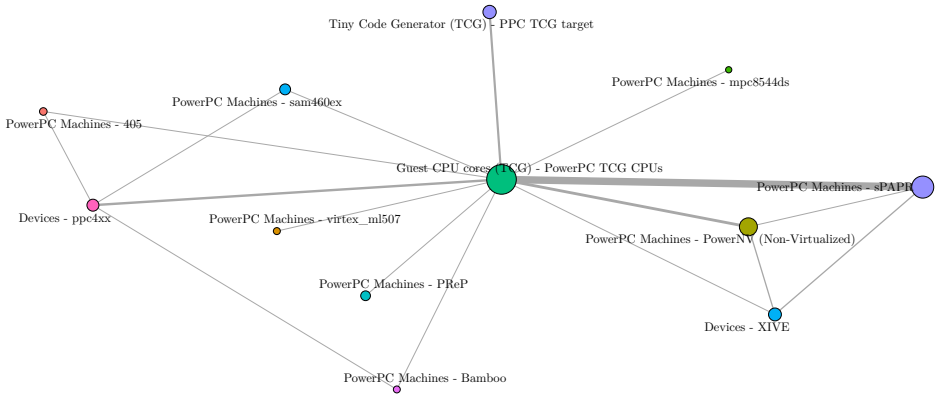


Figure 39: A network view for PowerPC specific sections.

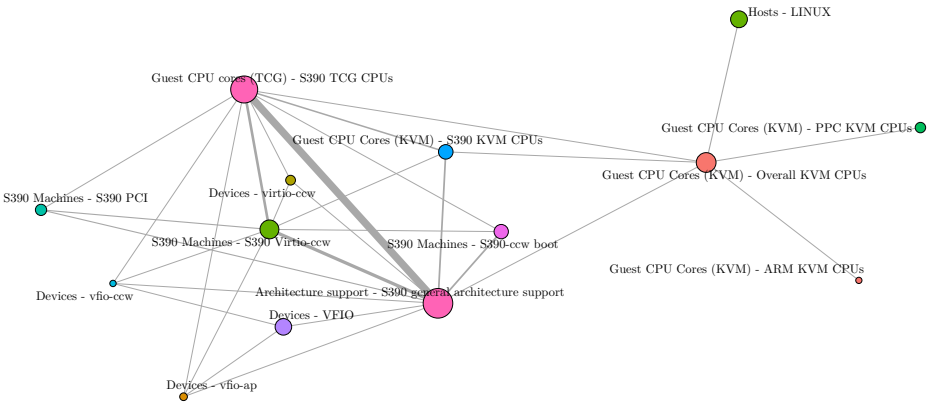


Figure 40: A network view for S390 specific sections.

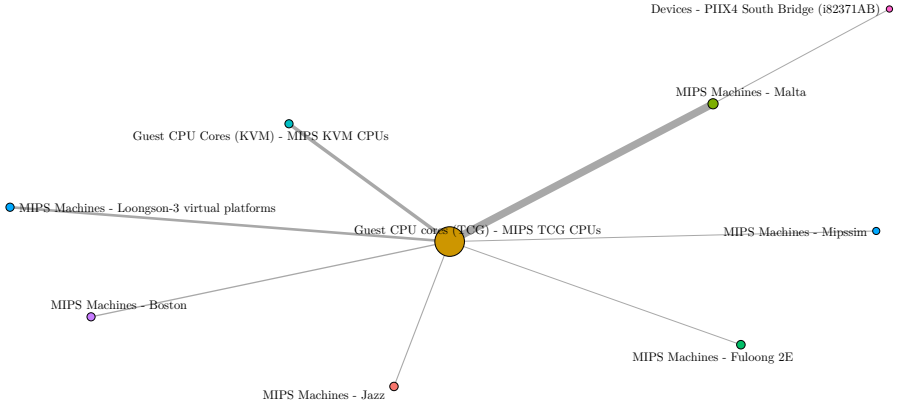


Figure 41: A network view for MIPS themed sections.

A.3. U-Boot

Here we present a few selected network views for U-Boot v2022.01.

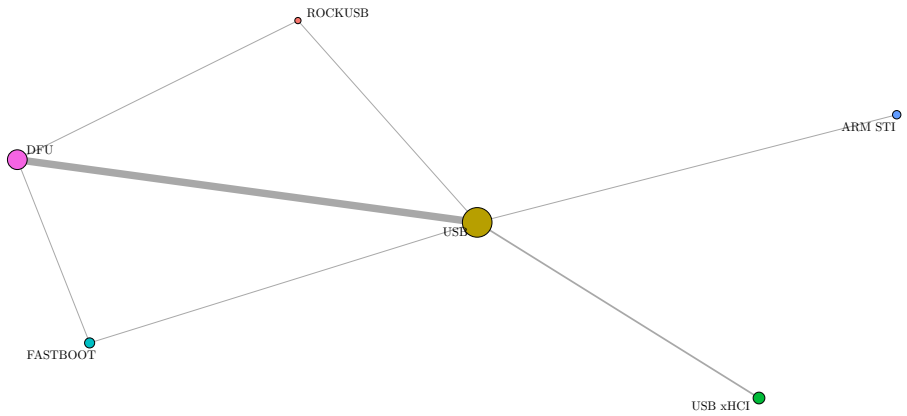


Figure 42: A USB themed network view.

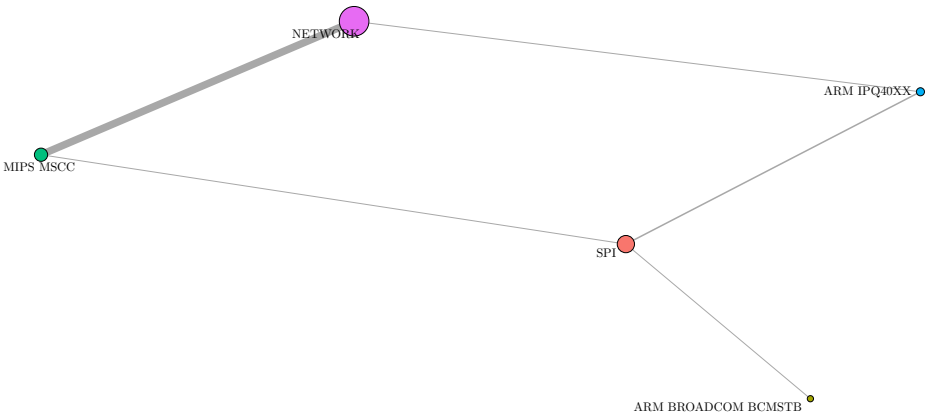


Figure 43: A network view for network themed sections.

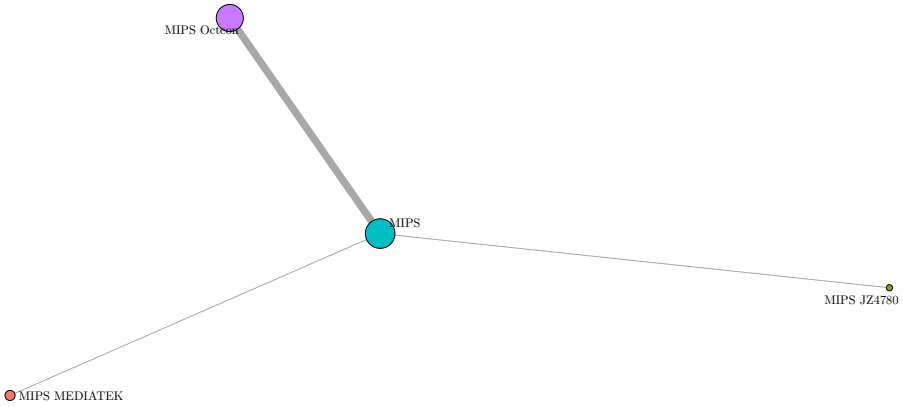


Figure 44: A network view for MIPS themed sections.

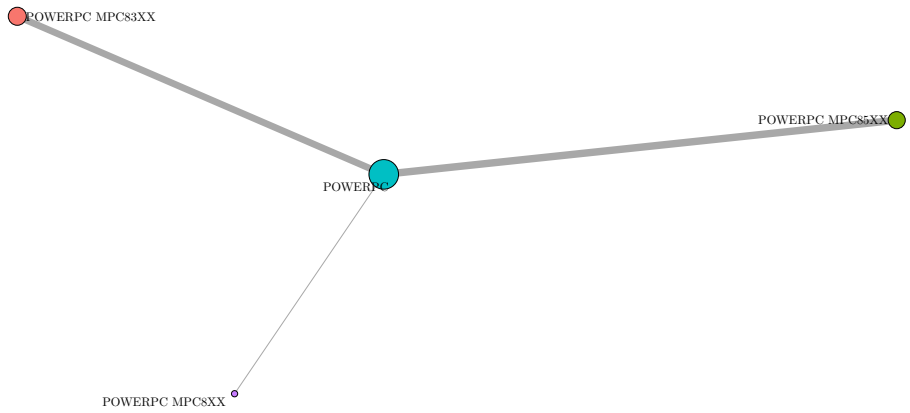


Figure 45: A network view for PowerPC themed sections.

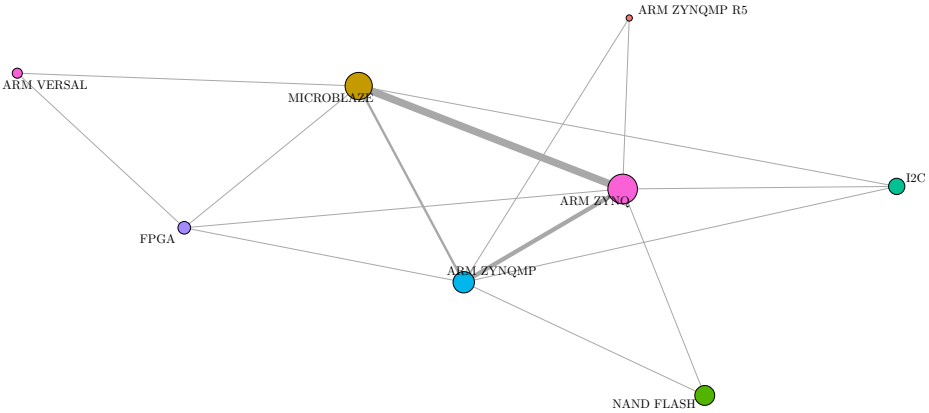


Figure 46: A network view for ARM themed sections.

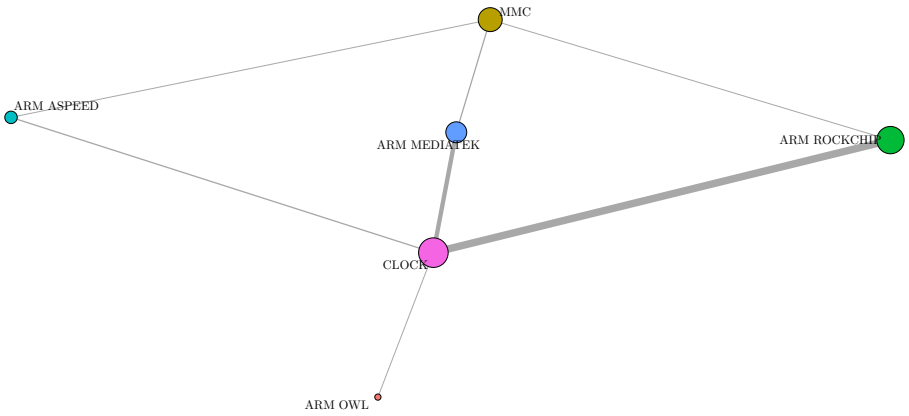


Figure 47: A network view for ARM and clock themed sections.

A.4. Xen

Here we present a few selected network views for Xen RELEASE-4.15.0.

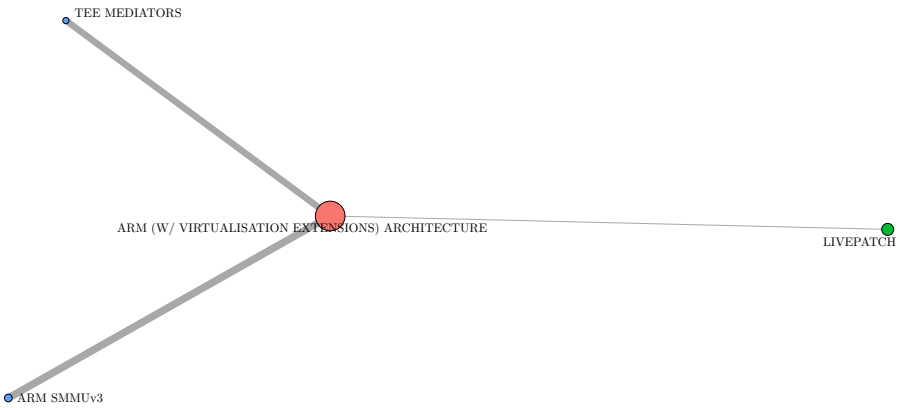


Figure 48: A network view for ARM virtualisation architecture.

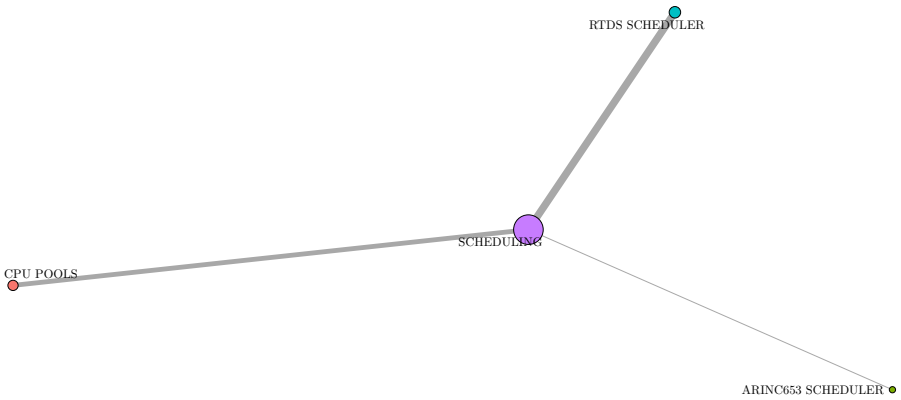


Figure 49: A network view for scheduling themed sections.

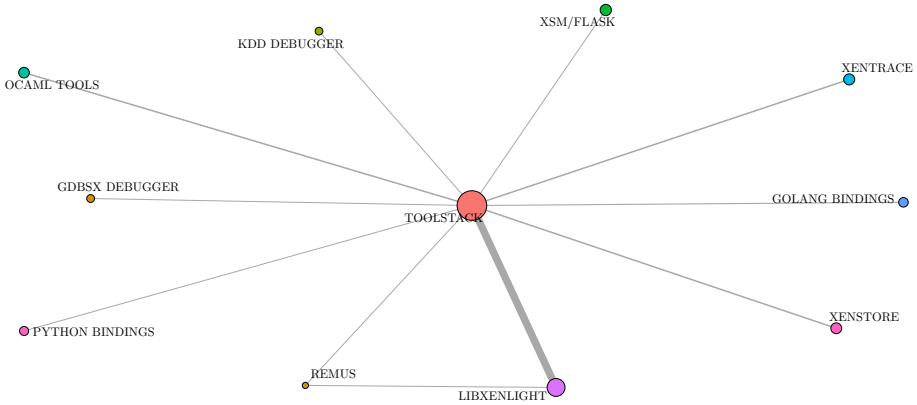


Figure 50: A network view for the general toolstack.

B. References

[1] Mark Aberdour. "Achieving quality in open-source software." In: *IEEE software* 24.1 (2007).

[2] *About Linux Kernel*. URL: `https://www.kernel.org/linux.html` (visited on 05/19/2022).

[3] Emad Aghajani et al. "Software Documentation Issues Unveiled." In: 2019.

[4] Adam Alami, Marisa Leavitt Cohn, and Andrzej Wąsowski. "Why Does Code Review Work for Open Source Software Communities?" In: 2019.

[5] Periklis Andritsos and Vassilios Tzerpos. "Information-theoretic software clustering." In: *IEEE Transactions on Software Engineering* 31.2 (2005).

[6] Nicolas Anquetil and Timothy C Lethbridge. "Experiments with clustering as a software remodularization method." In: IEEE. 1999.

[7] Nicolas Anquetil and Timothy C Lethbridge. "Recovering software architecture from the names of source files." In: *Journal of Software Maintenance: Research and Practice* 11.3 (1999).

[8] Sven Apel et al. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated, 2013. ISBN: 3642375200.

[9] Usman Ashraf et al. "Do Communities in Developer Interaction Networks align with Subsystem Developer Teams? An Empirical Study of Open Source Systems." In: *arXiv preprint arXiv:2104.03648* (2021).

[10] Association for Computing Machinery. *Artifact review and badging (version 1.1)*. URL: `https://www.acm.org/publications/policies/artifact-review-and-badging-current` (visited on 05/19/2022).

[11] Stefan Axelsson. "The Normalised Compression Distance as a file fragment classifier." In: *digital investigation* 7 (2010), S24–S31.

[12] Monya Baker. "Reproducibility crisis." In: *Nature* 533.26 (2016), pp. 353–66.

[13] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. 3rd. Addison-Wesley Professional, 2012.

[14] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. 46. Califor-nia, USA. 2005, pp. 10–5555.

[15] Dirk Beyer and Andreas Noack. "Clustering software artifacts based on frequent common changes." In: IEEE. 2005.

[16] Christian Bird et al. "Don't touch my code! Examining the effects of ownership on software quality." In: 2011.

[17] Vincent D Blondel et al. "Fast unfolding of communities in large networks." In: *Journal of statistical mechanics: theory and experiment* 2008.10 (2008).

[18] Carl Boettiger. "An introduction to Docker for reproducible research." In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 71–79.

[19] Ivan T Bowman and Richard C Holt. "Reconstructing ownership architectures to help understand software systems." In: IEEE. 1999.

[20] Thanh Bui. "Analysis of docker security." In: *arXiv preprint arXiv:1501.02967* (2015).

[21] Rolando P Reyes Ch, Oscar Dieste, Natalia Juristo, et al. "Statistical errors in software engineering experiments: A preliminary literature review." In: IEEE. 2018.

[22] R. Cilibrasi and P. M.B. Vitanyi. “Clustering by Compression.” In: *IEEE Trans. Inf. Theor.* 51.4 (2005).

[23] Rudi Cilibrasi. *Statistical inference through data compression*. University of Amsterdam, 2006.

[24] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. “Finding community structure in very large networks.” In: *Physical review E* 70 (2004).

[25] Jonathan Corbet. *MAINTAINERS truth and fiction*. `https://lwn.net/Articles/842415/`. Published at lwn.net. 2021.

[26] A. Courouble. *On History-aware Multi-activity Expertise Models*. Mémoire de maîtrise. École polytechnique de Montréal, 2018.

[27] Gabor Csardi. *Community structure via short random walks*. URL: `https://igraph.org/r/doc/cluster_walktrap.html` (visited on 09/06/2020).

[28] Gabor Csardi and Tamas Nepusz. “The igraph software package for complex network research.” In: *InterJournal* Complex Systems (2006).

[29] Edson Dias et al. “What Makes a Great Maintainer of Open Source Projects?” In: IEEE. 2021.

[30] Steve Easterbrook et al. “Selecting empirical methods for software engineering research.” In: Springer, 2008.

[31] Carolyn D. Egelman et al. “Predicting Developers’ Negative Feelings about Code Review.” In: 2020.

[32] Isabella Ferreira et al. “A Longitudinal Study on the Maintainers’ Sentiment of a Large Scale Open Source Ecosystem.” In: 2019.

[33] Santo Fortunato. “Community detection in graphs.” In: *Physics reports* (2010).

[34] Andrew Forward and Timothy C Lethbridge. “The relevance of software documentation, tools and technologies: a survey.” In: 2002.

[35] Edward B Fowlkes and Colin L Mallows. “A method for comparing two hierarchical clusterings.” In: *Journal of the American statistical association* 78.383 (1983).

[36] Alfonso Fuggetta. “Open source software—-an evaluation.” In: *Journal of Systems and software* 66.1 (2003).

[37] Andrea Gobbi et al. “Fast randomization of large genomic datasets while preserving alteration counts.” In: *Bioinformatics* 30.17 (2014).

[38] Kevin Gudeth et al. “Delivering secure applications on commercial mobile devices: the case for bare metal hypervisors.” In: *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. 2011, pp. 33–38.

[39] S. Guthals and P. Haack. *GitHub For Dummies*. Wiley, 2019.

[40] Mathew Hall, Neil Walkinshaw, and Phil McMinn. “Effectively Incorporating Expert Knowledge in Automated Software Remodularisation.” In: *IEEE Trans. Software Eng.* 44.7 (2018).

[41] Paul Halmos. *Naive Set Theory*. Van Nostrand, 1960.

[42] Nikolay Harutyunyan. “Corporate open source governance of software supply chains.” PhD thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2019.

[43] Ahmed E. Hassan. “The road ahead for Mining Software Repositories.” In: *2008 Frontiers of Software Maintenance*. 2008, pp. 48–57. DOI: `10.1109/FOSM.2008.4659248`.

[44] *IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*. International Electrotechnical Commission.

[45] Francesco Iorio et al. "Efficient randomization of biological networks while preserving functional characterization of individual nodes." In: *BMC bioinformatics* 17.1 (2016).

[46] *ISO 26262: Road vehicles – Functional safety*. International Organization for Standardization.

[47] Anil K. Jain. "Data Clustering: 50 Years beyond K-Means." In: *Pattern Recogn. Lett.* 31.8 (2010).

[48] Yujuan Jiang, Bram Adams, and Daniel M German. "Will my patch make it? and how fast? case study on the linux kernel." In: IEEE. 2013.

[49] Yujuan Jiang et al. "Tracing back the history of commits in low-tech reviewing environments: a case study of the linux kernel." In: 2014.

[50] Mira Kajko-Mattsson. "A survey of documentation practice within corrective maintenance." In: *Empirical Software Engineering* 10.1 (2005).

[51] Shuah Khan. *Advancing Open Source Safety-Critical Systems*. 2021.

[52] Timo Koponen and Virpi Hotti. "Open source software maintenance process framework." In: 2005.

[53] Greg Kroah-Hartman. "Why kernel development still uses email." In: *Linux Weekly News (LWN)* (2016). URL: `https://lwn.net/Articles/702177/`.

[54] Julia Lawall and Gilles Muller. "Coccinelle: 10 Years of Automated Evolution in the Linux Kernel." In: USENIX Association, 2018.

[55] Hannu Leppinen. "Current use of Linux in spacecraft flight software." In: *IEEE Aerospace and Electronic Systems Magazine* 32.10 (2017).

[56] Christian Lindig. "Mining Patterns and Violations Using Concept Analysis." In: Morgan Kaufmann / Elsevier, 2015.

[57] Rudi Lutz. "Recovering high-level structure of software systems using a minimum description length principle." In: Springer. 2002.

[58] Spiros Mancoridis et al. "Bunch: A clustering tool for the recovery and maintenance of software system structures." In: IEEE. 1999.

[59] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. John Wiley & Sons, 2010.

[60] Brian S Mitchell and Spiros Mancoridis. "On the automatic modularization of software systems using the bunch tool." In: *IEEE Transactions on Software Engineering* 32.3 (2006).

[61] Pablo Montero and José A. Vilar. "TSclust: An R Package for Time Series Clustering." In: *Journal of Statistical Software* 62.1 (2014).

[62] Hausi A. Müller et al. "A reverse-engineering approach to subsystem structure identification." In: *Journal of Software Maintenance: Research and Practice* 5.4 (1993).

[63] Mark EJ Newman and Michelle Girvan. "Finding and evaluating community structure in networks." In: *Physical review E* 69.2 (2004), p. 026113.

[64] Pierre Pluye and Quan Nha Hong. "Combining the power of stories and the power of numbers: mixed methods research and mixed studies reviews." In: *Annual review of public health* 35 (2014).

[65] Pascal Pons and Matthieu Latapy. “Computing communities in large networks using random walks.” In: Springer. 2005.

[66] Rachel Potvin and Josh Levenberg. “Why Google Stores Billions of Lines of Code in a Single Repository.” In: *Commun. ACM* 59.7 (2016).

[67] *Preempt-RT Wiki*. URL: `https://rt.wiki.kernel.org/` (visited on 05/19/2022).

[68] *QEMU Docs - Submitting a Patch*. URL: `https://www.qemu.org/docs/master/devel/submitting-a-patch.html` (visited on 05/19/2022).

[69] *QEMU Wiki - Main Page*. URL: `https://wiki.qemu.org/Main_Page` (visited on 05/19/2022).

[70] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2020.

[71] Md Tajmilur Rahman and Peter C Rigby. “Contrasting development and release stabilization work on the Linux kernel.” In: vol. 2014. 2014.

[72] Ralf Ramsauer, Daniel Lohmann, and Wolfgang Mauerer. “Observing custom software modifications: A quantitative approach of tracking the evolution of patch stacks.” In: 2016.

[73] Ralf Ramsauer, Daniel Lohmann, and Wolfgang Mauerer. “The list is the process: Reliable pre-integration tracking of commits on mailing lists.” In: IEEE. 2019.

[74] Ralf Ramsauer et al. “The Sound of Silence: Mining Security Vulnerabilities from Secret Integration Channels in Open-Source Projects.” In: CCSW’20. Association for Computing Machinery, 2020.

[75] Eric Raymond. “The cathedral and the bazaar.” In: *Knowledge, Technology & Policy* (1999).

[76] Peter C. Rigby and Margaret-Anne Storey. “Understanding Broadcast Based Peer Review on Open Source Software Projects.” In: ICSE ’11. New York, NY, USA: Association for Computing Machinery, 2011.

[77] Andrew Rosenberg and Julia Hirschberg. “V-measure: A conditional entropy-based external cluster evaluation measure.” In: 2007.

[78] Martin Rosvall and Carl T Bergstrom. “Maps of information flow reveal community structure in complex networks.” In: *arXiv preprint physics.soc-ph/0707.0609* (2007).

[79] Daniel Russo. “Benefits of open source software in defense environments.” In: Springer. 2016.

[80] Julio Sincero et al. “Is The Linux Kernel a Software Product Line?” In: *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*. Ed. by Frank van der Linden and Björn Lundell. Kyoto, Japan, 2007. URL: `http://fame-dbms.org/publications/SPLC-OSSPL2007-Sincero.pdf`.

[81] Robert R Sokal. “The principles and practice of numerical taxonomy.” In: *Taxon* (1963).

[82] Conrad Spiteri et al. “Real-time visual sinkage detection for planetary rovers.” In: *Robotics and Autonomous Systems* 72 (2015).

[83] *Submitting Xen Project Patches*. URL: `https://wiki.xenproject.org/wiki/Submitting_Xen_Project_Patches` (visited on 05/19/2022).

[84] Xin Tan. “Reducing the Workload of the Linux Kernel Maintainers: Multiple-Committer Model.” In: ESEC/FSE 2019. Association for Computing Machinery, 2019.

[85] Xin Tan, Minghui Zhou, and Brian Fitzgerald. "Scaling Open Source Communities: An Empirical Study of the Linux Kernel." In: 2020.

[86] The Kernel Community. *Submitting patches: the essential guide to getting your code into the kernel*. https://www.kernel.org/doc/html/latest/process/submitting-patches.html. 2020.

[87] The Linux Foundation. *The Linux Foundation Launches ELISA Project Enabling Linux In Safety-Critical Systems*. URL: https://www.linuxfoundation.org/press-release/the-linux-foundation-launches-elisa-project-enabling-linux-in-safety-critical-systems/ (visited on 05/19/2022).

[88] The Linux Foundation. *What Is Linux?* URL: https://www.linux.com/what-is-linux/ (visited on 05/19/2022).

[89] *The many and varied uses of QEMU*. URL: https://www.linaro.org/blog/many-uses-of-qemu/ (visited on 05/19/2022).

[90] *The Open Source Definition (Annotated)*. URL: https://opensource.org/docs/definition.php (visited on 05/19/2022).

[91] Paolo Tonella. "Concept analysis for module restructuring." In: *IEEE Transactions on software engineering* 27.4 (2001).

[92] *U-Boot Docs - Patches and Feature Requests*. URL: https://www.denx.de/wiki/U-Boot/Patches (visited on 05/19/2022).

[93] *U-Boot Documentation: README*. URL: https://source.denx.de/u-boot/u-boot/raw/HEAD/README (visited on 05/19/2022).

[94] Arie Van Deursen and Tobias Kuipers. "Identifying objects using cluster and concept analysis." In: 1999.

[95] Nguyen Xuan Vinh, Julien Epps, and James Bailey. "Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance." In: *The Journal of Machine Learning Research* 11 (2010).

[96] Huaiqing Wang and Chen Wang. "Open source software adoption: a status report." In: *IEEE Software* 18.2 (2001). DOI: 10.1109/52.914753.

[97] Titus Winters, Tom Manshreck, and Hyrum Wright. *Software Engineering at Google*. O'Reilly Media, Inc, 2020.

[98] *Xen Project Hypervisor 4.15 now Available*. URL: https://xenproject.org/2021/04/08/xen-project-hypervisor-4-15/ (visited on 05/19/2022).

[99] *Xen Wiki - Xen Project Software Overview*. URL: https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview (visited on 05/19/2022).

[100] Meng Yan et al. "Software quality assessment model: A systematic mapping study." In: *Science China Information Sciences* 62.9 (2019).

[101] Wayne W Zachary. "An information flow model for conflict and fission in small groups." In: *Journal of anthropological research* 33.4 (1977), pp. 452–473.

[102] Ying Zhao and George Karypis. "Criterion functions for document clustering: Experiments and analysis." In: (2001).

[103] Junji Zhi et al. "Cost, benefits and quality of software development documentation: A systematic mapping." In: *Journal of Systems and Software* 99 (2015).

[104] Minghui Zhou et al. "On the Scalability of Linux Kernel Maintainers' Work." In: ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017.

List of Figures

List of Tables

Selbstständigkeitserklärung

Ich habe die Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und bisher keiner anderen Prüfungsbehörde vorgelegt. Außerdem bestätige ich hiermit, dass die vorgelegten Druckexemplare und die vorgelegte elektronische Version der Arbeit identisch sind und dass ich von den in § 27 Abs. 6 vorgesehenen Rechtsfolgen Kenntnis habe.

Unterschrift: _____