Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Comparing the intensity of variability changes in software product line evolution[☆],[☆☆]

Christian Kröher[*], Lea Gerling, Klaus Schmid

University of Hildesheim, Institute of Computer Science, Universitätsplatz 1, 31141, Hildesheim, Germany

ARTICLE INFO

Article history:
Received 4 November 2022
Accepted 27 April 2023
Available online 13 May 2023

Keywords:
Software product line evolution
Evolution analysis
Variability changes
Intensity
Comparison

ABSTRACT

The evolution of a Software Product Line (SPL) typically affects multiple kinds of artifacts. The *intensity* (frequency and amount) in which developers change variability information in them was unknown, until we introduced a fine-grained approach for the variability-centric extraction and analysis of changes to code, build, and variability model artifacts. Its application to the commits of the Linux kernel revealed that changes to variability information occur infrequently and only affect small parts of the analyzed artifacts. Further, we outlined how these results may improve certain analysis and verification tasks during SPL evolution. However, the sole analysis of a single SPL did not allow for generic conclusions.

In this paper, we extend our previous work to a comparative evolution analysis of four SPLs. We provide a detailed analysis of the individual intensities of variability changes by applying our updated approach to each SPL. A comparison of these results confirms our findings of infrequent and small changes to variability information in our previous study. However, differences in the details of these changes exist, which we cannot explain solely by the characteristics of the SPLs or their development processes. We discuss their implications on supporting SPL evolution and on our previous optimization proposals.

1. Introduction

The evolution of a Software Product Line (SPL) typically affects a variety of artifact types, like source code, build system, and variability model artifacts (Passos et al., 2016; Hellebrand et al., 2014; Neves et al., 2015). For each type, changes can be differentiated into explicit changes to *variability information* and arbitrary changes to *artifact-specific information*. Variability information realizes the configuration knowledge. It enables the customization of artifacts for a specific product, like the configuration options in the variability model or explicit references to these options in pre-processor statements (Hunsen et al., 2016). Artifact-specific information defines the core content of an artifact as in traditional software development. Examples for this type of information are general program definitions in code artifacts, which includes code realizing particular features, or build process definitions in build artifacts.

A particular focus of SPL research is on understanding the evolution of the variability information. The results range from general SPL evolution scenarios (Neves et al., 2015; Sampaio et al., 2016) to the independent evolution of such information in specific types of artifacts (Dintzner et al., 2017; Lotufo et al., 2010), or their co-evolution (Hellebrand et al., 2014; Dintzner et al., 2016; Passos et al., 2016). However, these results are typically based on a feature-perspective, which considers changes in relation to a specific feature and, hence, abstracts from implementation details of SPL evolution. For example, it consolidates individual changes, e.g., introduced by several commits, and hides other changes, which affect features out of scope or artifact-specific information. Hence, the *frequency* with which developers generally change a specific *amount* of variability information in different types of artifacts independent from its relation to a specific feature was previously unknown.

In our previous work, we therefore presented a fine-grained approach to identify the *intensity* (the frequency and amount) of changes to code, build, and variability model artifacts (Kröher et al., 2018). Fig. 1 illustrates the two main processes of this approach. It performs a *Commit Extraction* from a *SPL Repository*. The *Extracted Commits* are input to a *Commit Analysis*, which provides the *Evolution Data* with the number of changed lines containing

[☆] This work is partially supported by the Evoline project, funded by the DFG (German Research Foundation) under Priority Programme SPP 1593 and by the ITEA3 project REVaMP funded by the BMBF (German Ministry of Research and Education) under grant 01IS16042H. Any opinions expressed herein are solely by the authors and not of the DFG or BMBF.

[☆☆] Editor: Laurence Duchien.

[*] Corresponding author.
E-mail addresses: kroeher@sse.uni-hildesheim.de (C. Kröher), gerling@sse.uni-hildesheim.de (L. Gerling), schmid@sse.uni-hildesheim.de (K. Schmid).

https://doi.org/10.1016/j.jss.2023.111737
0164-1212/© 2023 Elsevier Inc. All rights reserved.

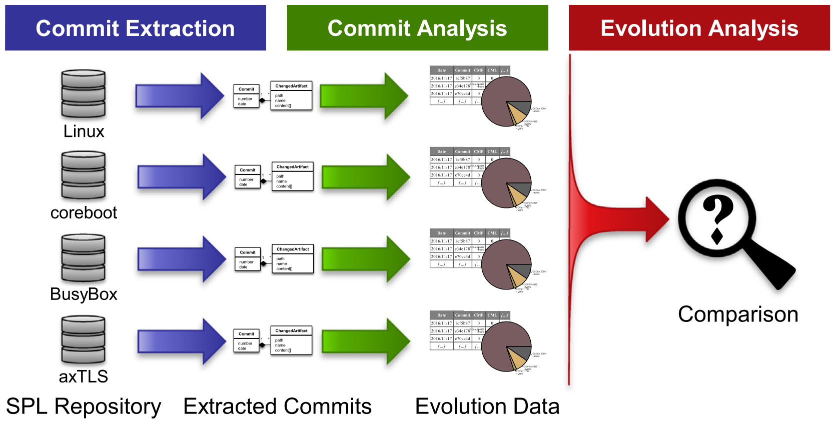


Fig. 1. Analysis approach.

artifact-specific and variability information in the three types of artifacts for each commit. The application of these processes to the Linux kernel revealed that changes to variability information occur infrequently and only affect small parts of the analyzed artifacts. These results built the foundation for our approach to incremental SPL verification (Kröher et al., 2022a). We exploited the knowledge from our evolution analysis to significantly accelerate the well-established dead code analysis (Tartler et al., 2011; Nadi and Holt, 2012; Dietrich et al., 2012) and reduce re-analysis effort from 17 min to 3 s on median per commit. However, all these results stem from an exclusive usage of the Linux kernel as case study. This limitation has a notable impact on the generosity of our findings and improvements. In particular, drawing conclusions for SPL evolution (support) in general was not possible.

In this paper, we aim at a generalization of our previous results by applying our approach from Kröher et al. (2018) to three additional SPLs. Fig. 1 illustrates this extension. It presents as our case studies the *coreboot* firmware (coreboot team, 2022), *BusyBox* UNIX utilities (BusyBox team, 2022), and *axTLS* embedded SSL (Rich, 2022) besides the *Linux kernel* (The kernel development community, 2022c), which we already discussed in our previous work (Kröher et al., 2018). For each SPL, the *Commit Extraction* enables the *Commit Analysis* to provide their individual *Evolution Data* as for the Linux kernel before exclusively. In the *Evolution Analysis*, we compare these individual results to answer the following research questions:

RQ1 How much does the intensity of changes of variability information vary between different SPLs in general?
RQ2 How much does the intensity of changes of variability information in code, build, and variability model artifacts vary between different SPLs?
RQ3 How many code, build, or variability model lines containing variability information are changed in different SPLs on average?

The SPLs selected as case studies enable a review of our previous conclusions and support our goal of generalizing them. For example, the diversity of their evolution in the scope of this paper ranges from about 1 million commits in more than 17 years (Linux kernel), over about 8000 commits in almost 8 years (BusyBox), to 179 commits in 12 years of (axTLS). We present this and other differences regarding project sizes and development efforts in more detail as part of explaining our analysis setup. Hence, we make the following contributions:

- We describe an updated version of our approach for the automated extraction and analysis of changed lines from commits as well as their differentiation into artifact-specific and variability changes; the update consists of a generalization of the main processes and a dedicated description of their realization including their customization for different SPLs.
- We reveal and analyze the evolution of artifact-specific and variability information in code, build, and variability model artifacts in four SPLs.
- We compare and discuss this evolution data based on our research questions to provide a broader understanding on SPL evolution.

In addition, we review our previous proposals for exploiting the results from analyzing the intensity of variability changes of the Linux kernel exclusively. This includes a critical consideration of our performance results in Kröher et al. (2022a) with respect to our extended evolution analysis.

In the next section, we discuss related work. Section 3 explains the variability realization in our case studies as a background for understanding our approach presented in Section 4. Further, Section 3 outlines our previous results from analyzing SPL evolution in Kröher et al. (2018) as additional context and motivation for this paper. Section 5 describes the technical realization of our approach. In Section 6, we define the setup (hardware, software, data set), the execution, and the validation of our results. Section 7 presents the individual evolution data from the application of our approach to the case studies. Section 8 compares these individual results for a general discussion in accordance with our research questions. We discuss threats to validity in Section 9 and provide our conclusions in the final section.

2. Related work

In this paper, we analyze the evolution of artifact-specific and variability information in code, build, and variability model artifacts of four different SPLs. In particular, we investigate the intensity of changes to variability information in these artifact types using a tool-set for extracting and analyzing the commits of the respective SPL repositories. Further, we compare the individual evolution data to draw general conclusions about the intensity with which developers change variability information in practice. Hence, we discuss related work, which also focuses on understanding the evolution of variability in SPLs as well as extracting and analyzing evolution data from SPL repositories. We start this discussion with related work on **multiple case studies** as in this paper. We then proceed with related work, which considers a **single case study** only, sharing that main difference to our contribution. Finally, we delimit our work from related **tooling** for revealing evolution data.

Variability evolution in multiple case studies. Oliveira et al. investigate the evolution of feature dependencies in 15 open-source projects to identify their most common types of changes (Oliveira et al., 2019). The explorative study comprises 918 releases over all projects for which the authors identify and compare the dependencies in the preprocessor-based C-code. Our analysis considers additional variability model and build artifacts on the level of individual commits instead of entire releases. Further, we rely neither on a feature-perspective nor on (code) semantics as required to identify feature dependencies.

Michelon et al. analyze the lifecycle of features in four highly-configurable software systems (Michelon et al., 2021). Their particular goal is to reveal when and how developers introduce, revise, and remove features in the source code. The authors therefore consider changes to preprocessor statements in individual commits similar to our approach. However, our analysis counts the individual line changes independent of particular features for different types of artifacts, while Michelon et al. further process the commits to trace feature definitions in code exclusively.

Neves et al. provide a set of safe evolution templates based on a pair-wise commit analysis of two SPLs (Neves et al., 2015). That paper is an extension of their previous work on such templates, which relies on the same case studies (Neves et al., 2011). The analysis in this paper does not aim at the derivation of any (safe) evolution templates. Hence, we do not check particular changes to potentially different types of artifacts for correctness. Further, we do not rely on a specific approach to SPL evolution, like the product line refinement theory defined in Borba et al. (2012).

Chaikalis et al. investigate the effect of evolution on feature scattering in code artifacts in four open-source Java systems (Chaikalis et al., 2015). Their particular focus is on the support for analyzing and visualizing of data concerning the evolution of that scattering. In contrast, we present the analysis results and their visualization for the changes to variability and artifact-specific information across different types of artifacts based on a commit-wise analysis of four SPLs. Further, we do not investigate the locations of these changes in the file systems.

Svahnberg and Bosch evaluate different releases of two industrial case studies to provide evolution categories for requirements, architecture, and components along with supporting guidelines (Svahnberg and Bosch, 1999). On the one hand, we do not consider changes to artifacts, like requirements or other documents. On the other hand, we focus on changed lines rather than on the entire architecture or components.

Variability evolution in a single case study. Passos et al. extract four evolution patterns from three release pairs of the Linux kernel to explain the variability co-evolution in code, build, and variability model artifacts (Passos et al., 2012). The authors further refine and extend this initial pattern catalog over time based on larger samples of the Linux kernel history (Passos et al., 2013, 2016). Their core contribution is to provide a better understanding of how SPLs evolve, which is in line with our contribution in this paper. However, their focus is on detecting the addition or removal of a feature in the variability model first and then tracking changes to related build and code artifacts in previous or subsequent commits. We detect changes to artifact-specific and variability information for each commit instead of tracking feature changes over commit sequences. Further, we are interested in the intensity of changes to these types of information and their distribution across different types of artifacts rather than deriving co-evolution patterns.

Sampaio et al. provide a set of partially safe refinement templates from the Linux kernel (Sampaio et al., 2016). These templates complement the initial set of completely safe ones by Neves et al. (2015, 2011) mentioned above. Finally, Gomes et al. investigate the frequency with which safe and partially safe templates occur in SPL evolution using another single open-source project as case study (Gomes et al., 2019). As the work by Sampaio et al. and Gomes et al. stem from the work by Neves et al. the same differences to our contribution exist.

Zhang et al. analyze 31 versions of an industrial product line to derive a set of metrics for detecting variability erosion in code artifacts (Zhang et al., 2013). Hellebrand et al. extend this work towards six metrics based on monitoring the co-evolution of the variability model and code artifacts over eleven versions of the same industrial SPL (Hellebrand et al., 2014). Both approaches rely on investigating entire versions, which typically include multiple changes introduced by a set of commits. Hence, their approaches are more coarse-grained than our commit-based analysis. Further, Hellebrand et al. do not consider build artifacts, while Zhang et al. do additionally exclude the variability model as another source of variability information. The proposed metrics aim at sustaining productivity in SPL, while we are interested in supporting the general understanding of SPL evolution.

Lotufo et al. analyze exclusively the evolution of the Linux kernel variability model over 21 minor releases (Lotufo et al., 2010). The authors use this model and its evolution to identify operations on and refactorings to feature models as well as difficulties for humans to reason about model constraints. Besides the exclusive focus on variability model evolution, the major difference to our work is that the authors interpret the changes to come up with a set of typical operations.

Holdschick presents a collection of change cases from the automotive domain (Holdschick, 2012). As the analyzed SPL is realized using a model-driven approach, he focuses on the co-evolution of a functional model representing the components and their relations as well as a variant model, which consists of a feature model and the corresponding mapping between the features and their implementations. The author also aims towards a better understanding of SPL evolution, but his focus is on understanding specific types of changes and the necessary adaptions to the variant model. Our approach is different, as we analyze individual changes with a focus on the intensity of changes to variability information in code, build and variability model artifacts.

Adams et al. investigate the evolution of the Linux kernel build system to detect idioms and patterns in the dynamic build behavior (Adams et al., 2007). In contrast to other analyses, the authors use 14 early versions of Linux (from version 0.01 in 1991 to version 2.6.21.5 in 2007). Besides their exclusive focus on the build system artifacts, the authors present the evolution of code, build, variability model, and other artifacts in terms of the number of source lines of code. This complements our results as our analysis starts with the first Linux kernel commit of the Git repository in 2005, which is version 2.6.12-rc2. However, the authors do not present their numbers in terms of changed lines but as the total number of available lines for each artifact type per version.

Godfrey and Tu examine growth patterns of individual subsystems and the overall Linux kernel to gain an understanding of how and why the system as a whole evolves (Godfrey and Tu, 2000). The authors analyze 96 kernel versions including 34 stable releases and 62 development releases. Besides system growth, Israeli and Feitelson measure the complexity per function and their distribution (Israeli and Feitelson, 2010). The metrics are applied to 800 versions of the Linux kernel to observe effects like the reduction of complexity of the initial version code. Our focus is on the intensity of variability information changes and not on the growth of the system or the complexity and its distribution over time.

Tooling. Dintzner et al. present the FEVER tool for the automatic extraction of feature changes from commits (Dintzner et al., 2016). It provides detailed information about the evolution of individual features including any edits to the variability model, code, or build artifacts. The resulting timelines allow determining how and where the evolution of a specific feature takes place. In contrast, we do not track the evolution of individual features, but identify the number of changed lines containing artifact-specific and variability information in general.

3. Background

Identifying the intensity of variability changes in SPL evolution requires an understanding of the types of artifacts realizing this variability, their relevant content and relations. Hence, Section 3.1 explains the variability realization in the scope of this paper across code, build, and variability model artifacts. This part of the background provides necessary information to understand our analysis concepts presented in Section 4. The second part summarizes previous results from analyzing SPL evolution. In particular, Section 3.2 focuses on their limitations, which motivate the extension in this paper.

3.1. Variability realization

The SPLs we use as case studies likewise employ Kbuild (The kernel development community, 2022b) to realize variability. Hence, we select one representative of this application for explanation. Fig. 2 shows a real-world example of the Linux kernel.[1] It contains a code artifact in the upper part, two build artifacts in the lower part, and a variability model artifact in the middle. Further, arrows indicate relations between these artifacts as established by the Kbuild variability mechanism.

The source code in our scope consists of `*.c`- and `*.h`-files containing C-code and `*.S`-files containing assembler code. The presented `*.h`-file in Fig. 2 therefore is a representative for code artifacts in general in this paper (the following descriptions apply to all three file types). Variability in the source code is mainly realized by preprocessor statements (Hunsen et al., 2016). Lines 780, 782, and 784 in `perf_event.h` contain such statements, which control the presence of the enclosed C-code-fragments (lines 781 and 783). The decision of which fragment will be part of the compiled variant depends on the evaluation of the `#ifdef`-statement and the referenced symbol `CONFIG_X86_32` in line 780. This symbol is not defined in the source code, which is indicated by the prefix `CONFIG_`. This prefix differentiates symbols defined in the source code from configuration options defined in the variability model. Besides their main usage in preprocessor statements, some runtime code also references configuration options of the variability model, like in value assignments or as part of conditions and loops.

The `Kconfig`-file in Fig. 2 shows the definition for the `CONFIG_X86_32` symbol using the Kconfig language (The kernel development community, 2022a). In the considered SPLs multiple `Kconfig`-files exist, that together represent the variability model. In such a file, the keyword `config` in lines 9 and 1811 indicates the definition of a configuration option followed by its name: `X86_32` and `KEXEC_FILE`. These names do not include the prefix `CONFIG_`, which is only used to indicate references to configuration options in other artifacts. There are different types of configuration options, for example `X86_32` and `KEXEC_FILE` are both Boolean options. Further, configuration options can have dependencies, like the `depends on` relation described in line 11. It is also possible to connect dependencies with Boolean expressions. Further, Kconfig supports the definition of help texts and descriptions (e.g., lines 1818ff) for configuration options. They guide the end-user during the configuration of a variant, e.g., to understand the purpose and impact of selecting an option.

The selection or deselection of configuration options in the variability model leads to an adaptation of other artifacts. For example, if we want to configure a 32 bit kernel for a x86-architecture, the `X86_32` option has to be selected. This leads to the presence of line 781 in `perf_event.h`. The deselection of `X86_32` in turn leads to the presence of the code in the `#else`-block (line 783).

The `KEXEC_FILE` option in our example is used to adapt the build process, which is mostly implemented in `Kbuild`-files and `Makefiles`. The core content of these files consists of GNU Make commands (The kernel development community, 2022d). In the `Makefile` the reference to the configuration option (again indicated by the prefix `CONFIG_`) is part of a condition, which restricts the presence of the enclosed command in line 195 (Nadi and Holt, 2014). Thus, the selection of `KEXEC_FILE` leads to the execution of this command, while the deselection forces this command to be ignored. The `Kbuild`-file illustrates a different way to adapt the build process: the `KEXEC_FILE` option is used to modify the name of the variable `obj-`. The selection of this configuration option changes the name of the variable to `obj-y`. Hence, all files in the directory `purgatory` are compiled and linked during the build process. The deselection in turn leads to an exclusion of these files from the final variant.

The example from the Linux kernel illustrates the basic variability realization using Kbuild. However, some file names differ across our individual case studies in this paper. For example, variability model artifacts may be called `Config.*` instead of `Kconfig`. Similar changes in file names exist for build artifacts. While these name changes affect the identification and categorization of files during the analysis, they do not influence the respective variability information as explained in this section.

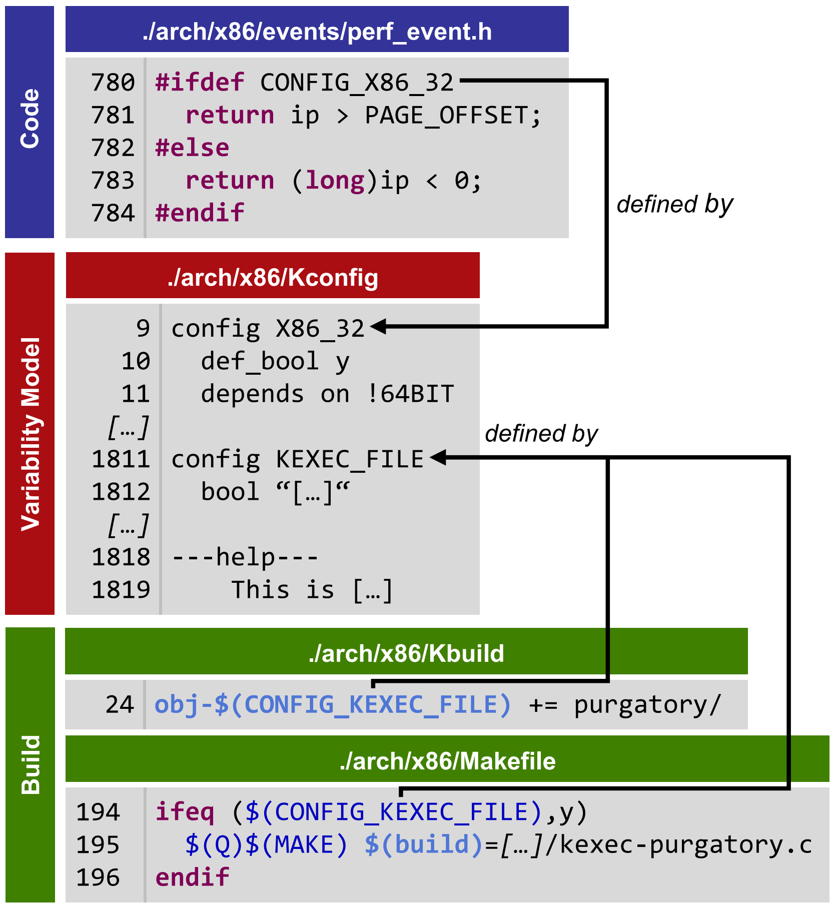
3.2. Previous results

The main motivation for the extended evolution analysis presented in this paper is based on the results of our previous analysis and, in particular, their limitations (Kröher et al., 2018; Kröher and Schmid, 2017b,a). Here, we rely on the same approach, which we will present in the next section. Its application to over 12 years of active Linux kernel development revealed that variability information does not change significantly relative to the total number of changes. The results showed that only 11% of the considered 662.110 commits change variability information in general. Further, if variability changes occur, they mostly apply to code artifacts, followed by variability model artifacts and build artifacts. These changes typically affect 1–10 lines independent of the specific type of artifacts.

The observations of our exclusive evolution analysis of the Linux kernel led us to an outline of exploitation opportunities (Kröher et al., 2018). A particular focus of these opportunities was on the group of family-based static SPL verification approaches (Thüm et al., 2014), like dead code (Tartler et al., 2011), feature effect (Nadi et al., 2015), and configuration mismatch (El-Sharkawy et al., 2017). We identified the potential of avoiding verification in about 89% of all commits in the evolution analysis completely, while reducing the verification effort for changed relevant artifacts to the affected few lines. Our subsequent performance analysis with an incremental approach to SPL verification proved these expected improvements (Kröher et al., 2022a). The fastest incremental strategy based on the algorithms for identifying variability changes in our evolution analysis reduced the dead code analysis from 17 min to 3 s in the best case.

An obvious limitation of the results outlined above is the exclusive consideration of the Linux kernel. While the SPL research community extensively uses it as a prominent reference

[1] Linux kernel version 4.8-rc1: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=29b4817.



```
#ifdef CONFIG_X86_32
  return ip > PAGE_OFFSET;
#else
  return (long)ip < 0;
#endif
```

```
config X86_32
  def_bool y
  depends on !64BIT
[…]
config KEXEC_FILE
  bool “[…]“
[…]
---help---
    This is […]
```

```
obj-$(CONFIG_KEXEC_FILE) += purgatory/
```

```
ifeq ($(CONFIG_KEXEC_FILE),y)
  $(Q)$(MAKE) $(build)=[…]/kexec-purgatory.c
endif
```

Fig. 2. Variability realization example from the Linux kernel.

example for a large-scale, real-world SPL, it is not a blueprint for all kinds of (highly-)configurable software. In particular, if we want to understand SPL evolution in general, the consideration of such projects with different processes and varying complexity is mandatory. For example, we assume different intensities of changes for SPLs, which are not as mature as the Linux kernel or rely on smaller teams with different coding- and commit-conventions. Hence, we apply the same evolution analysis to multiple SPLs in this paper and compare the results.

4. Approach

This section introduces our fine-grained, variability-centric approach to extract and analyze evolution data from SPL repositories. We first describe the commit extraction process in Section 4.1, which provides the set of commits and their changes in a structured way. Section 4.2 then explains the commit analysis process. For each extracted commit, this process identifies and classifies line changes introduced by developers in relevant code, build, and variability model artifacts. Hence, we also discuss the necessary concepts for identifying and classifying changes as part of the respective steps of that process. Further, we provide an example commit introducing changes to the artifacts described in Section 3.1 to illustrate the classification of individual line changes. This description of our approach avoids technical details for individual version control systems, like Git or SVN. However, we explain the processes in terms of abstract steps to enable their individual mapping to specific commands in Section 5.

4.1. *Commit extraction*

The process of extracting a single commit from a repository consists of two fundamental steps, which we further refine as illustrated in Fig. 3. The first step of the commit extraction collects the *commit header* information. This information defines the basic properties of a commit and, in particular, the corresponding commit number as well as the commit date. While the commit number is a unique identifier for each commit, we require the commit date to preserve the historically correct order of changes during the analysis. Step 1.1 in Fig. 3 first obtains the entire commit header information from a repository and identifies the number and the date. Next, step 1.2 adds this information to a new commit instance of the commit data model.

The second step extracts the *commit body*, which contains the actual changes the commit introduces to individual artifacts. Step 2.1 in Fig. 3 obtains this commit body in terms of a list of changed artifacts. For each changed artifact, the provided information consists of two fundamental blocks: the general artifact information, like the respective file path and name, as well as the actual changes to its content. These changes are available as lines marked with a leading “+” for additions and a leading “−” for deletions. In particular, step 2.1 always provides the entire file content to ensure that the commit analysis is able to find any statement necessary to classify a change clearly. Step 2.2 adds each changed artifact as a respective, individual instance of the commit data model with the path, name, and full content to the commit created in the first step.

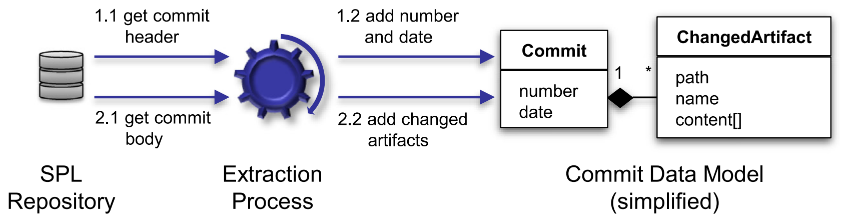


Fig. 3. Commit extraction steps.

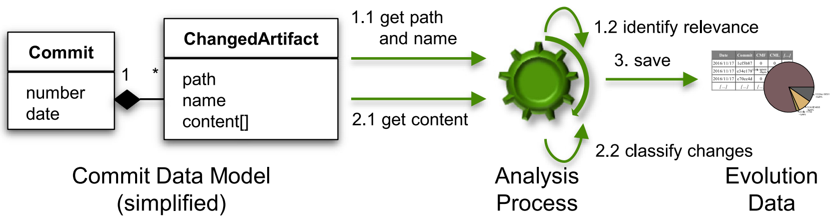


Fig. 4. Commit analysis steps.

4.2. *Commit analysis*

The analysis of a single commit provides its numbers of changed lines containing artifact-specific and variability information in code, build, and variability model artifacts. This process relies on the respective commit data (model) provided by the commit extraction as illustrated in Fig. 4. For each changed artifact of a commit instance, the analysis applies its two fundamental steps: *identifying the relevance* of a changed artifact for the goal of this approach and *classifying the changes*, if an artifact is relevant. Hence, we detail these steps and their concepts in the remainder of this section individually. Further, we refer to the final third step in Fig. 4 to the extent necessary to explain how we handle certain results for our evolution data.

4.2.1. *Identification*

The identification step defines a changed artifact to be relevant or not for our evolution analysis. It therefore requires the path and name of the respective artifact, which step 1.1 in Fig. 4 obtains form the commit data (model). The actual identification in step 1.2 determines the file type based on the name of the changed artifact. For example, we use regular expressions to match the name of a changed artifact against the file types described in Section 3.1 for the Linux kernel. Different expressions can be used for other SPLs, which we explain as part of the technical realization in Section 5. In general, this matching restricts the subsequent classification of changes to artifacts in the focus of this paper. Further, step 1.2 excludes changes from the analysis, which we categorize into one of the following Change Exclusion Cases (CEC):

CEC1 The changes only affect the file permissions, e.g., adding, deleting, or changing the permission of a file, but not the actual content of that file. Hence, step 3 in Fig. 4 saves the entire commit to a dedicated text file of the evolution data for manual inspection. For example, we used this file to detect false-negative classifications of changes.

CEC2 The commit does not contain changed artifacts, which is the case, e.g., if it actually describes a merge. Merge commits introduce the changes made by commits in a specific branch into another one. As we consider all commits in the analysis, the commits of a particular branch are processed anyway. Hence analyzing these changes again as part of a merge commit would identify the same changes twice. Step 3 in Fig. 4 saves these commits in the same way and for the same purpose as in (CEC1).

CEC3 We explicitly exclude some directories and file types, like documentation and script directories or plain text files. Step 1.2 skips these changed artifacts to avoid including documentation changes as changes affecting the actual variability of a SPL.

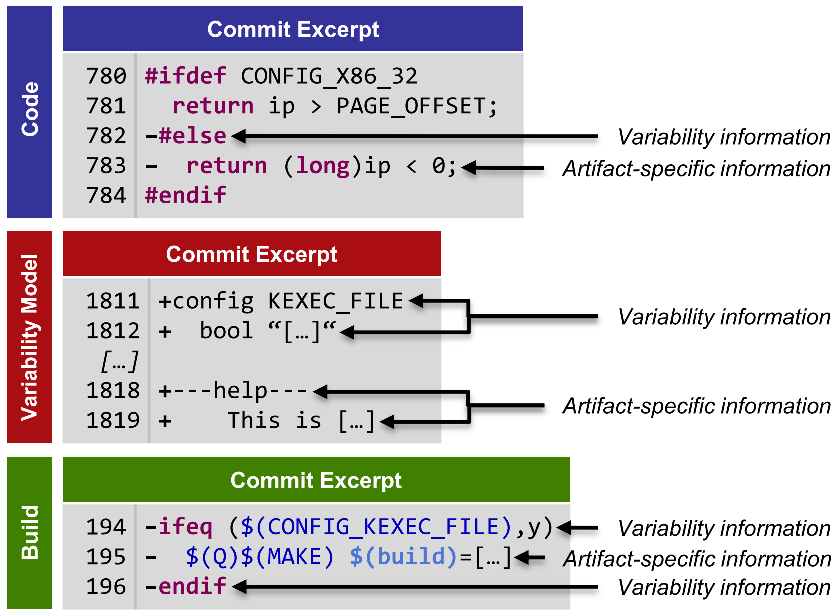
The result of the general identification step is a reduced set of changed artifacts for the classification step. In particular, we apply this identification to consider changed code, build, and variability model artifacts only. The next step then focuses on the individual analysis of changes in these artifacts.

4.2.2. *Classification*

The classification step considers individual line changes in each changed artifact identified as relevant for our analysis in the previous step. Hence, step 2.1 in Fig. 4 obtains the content of a changed artifact. It contains the entire file content with change markers indicating added and deleted lines (cf. Section 4.1). Fig. 5 illustrates such contents as commit excerpts from changed code, variability model, and build artifacts. We use Fig. 5 to explain step 2.2 in Fig. 4 and its classification into variability and artifact-specific information changes for each of these artifact types individually.

The upper part of Fig. 5 shows *code changes* and some context information in the `perf_event.h`-file from Fig. 2. The commit deletes the `#else`-block in lines 782 and 783 indicated by the leading “-”. The deletion of the `#else`-statement in line 782 is classified as a change to variability information. This is due to the `#ifdef`-statement in line 780, which marks the beginning of the entire preprocessor block. This `#ifdef`-statement references a configuration option, as explained in Section 3.1. Changes to preprocessor statements as well as general program statements, like variable definitions and control structures, are classified as changes to variability information, if they reference a configuration option somewhere in their whole structure. All other changes are classified as changes to artifact-specific information in code artifacts. Further, the classification differentiates changes to empty lines and comments. These changes are ignored for all artifact types, as they influence neither the variability nor the general program definition.

The lower part of Fig. 5 shows an example for *build changes* in the `Makefile` from Fig. 2. The illustrated change deletes the entire conditional block, indicated by the leading “-”. The surrounding control-statements (lines 194 and 196) are classified as changes to variability information, while the deletion of the



```
780 #ifdef CONFIG_X86_32
781   return ip > PAGE_OFFSET;
782 -#else
783 -   return (long)ip < 0;
784 #endif
```

```
1811 +config KEXEC_FILE
1812 +   bool "[…]"
 […]
1818 +---help---
1819 +     This is […]
```

```
194 -ifeq ($(CONFIG_KEXEC_FILE),y)
195 -   $(Q)$(MAKE) $(build)=[…]
196 -endif
```

Fig. 5. Change classification.

surrounded Make command in line 195 is classified as a change to artifact-specific information. The categorization of the `endif`-statement depends on the corresponding `ifeq`-statement, which references a configuration option. In general, any change to a Make command referencing a configuration option is categorized as a variability change. In particular, this covers statements like the adaptable `obj-` variable in Fig. 2. Other changes are classified as changes to artifact-specific information.

The middle part of Fig. 5 illustrates *variability model changes* in the `Kconfig`-file from Fig. 2 and the resulting categorization. In contrast to code and build artifacts, which mostly consist of artifact-specific information, the core content of variability model artifacts is variability information. In Fig. 5, the leading "+" indicates the addition of the `KEXEC_FILE` option in lines 1811 and following. Each added line related to the definition of the configuration option and its relation to other options (lines 1811 to 1817 in Fig. 5) is categorized as a change to variability information as they define the purpose and usage of the specific option. The added help text (lines 1818ff) is categorized as change to artifact-specific information, because these texts only support end-users during product configuration, but do not affect the variability. Hence, we consider all changes to help text in variability model artifacts as artifact-specific information changes, while all other changes are variability information changes.

5. Realization

The approach introduced in Section 4 was implemented in Java as a command line tool. It builds on the concept of experimentation workbenches (Schmid et al., 2019). The resulting Commit Analysis Infrastructure (ComAnI) (Kröher, 2022) therefore supports the execution of the same analysis for different SPLs by switching some configuration parameters. In particular, it explicitly separates the two main processes of our approach by individual extractor and analyzer plug-ins as illustrated in Fig. 6. This separation enables the exchange of an extractor depending on the version control system of the target SPL repository, while the analyzer stays the same. A configuration file defines the setup of the infrastructure and the selected plug-ins as well as the input and output properties. Hence, this file documents the main properties of each ComAnI execution, which significantly supports reproducibility of (individual parts of) our analysis described in Section 6. In this section, we focus on the main capabilities of ComAnI, the plug-ins we use in our analysis, and the relevant configuration parameters for it. We structure this explanation along Fig. 6 from left to right following the typical workflow of our approach and, hence, its realization. For a complete description of the open source tool ComAnI and its associated plug-ins, we refer to their public documentation (Kröher, 2022).

A *commit extractor* is an exchangeable plug-in. It instantiates the general commit extraction process (cf. Section 4.1) for a particular version control system. Each extractor requires the path to a local copy of a *SPL repository* from which it extracts the commits As part of this extraction, it translates the commit information from a specific version control system into instances of the generic *commit data model*. In general, commit extractors support the following variants of this extraction:

- A *full repository extraction* extracting all commits from a repository
- A *partial repository extraction* based on an additional list of commits to extract exclusively from a repository

We implemented both extraction variants for two specific commit extractors: one for Git and another one for SVN repositories. This allows us to use different commands in different orders to realize the steps of the general extraction process for a particular SPL repository. Table 1 provides the mapping between the relevant extraction steps and the specific commands for each version control system. Further, the *pre*-step in Table 1 defines the command to retrieve the list of all available commits in a repository as a *pre*requisite for a *full repository extraction*. These prerequisite commands do not require any additional parameters. In contrast, the commands for step 1.1 and 2.1 each assume a commit number as last parameter for which they return the following information:

- `git show -s --format=%ci` provides the committer date; it defines the point in time the changes become part of the respective repository (Chacon and Straub, 2014), which is required to preserve the historically correct order of changes during the analysis

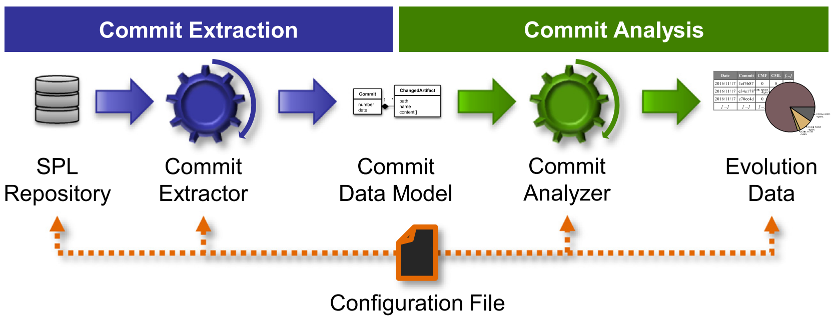


Fig. 6. Commit analysis infrastructure.

Table 1
Mapping of extraction steps to version control system commands.

		Commit extractor	
		Git	*SVN*
Extraction step (cf. Section 4.1)	*pre*	`git log --pretty=format:%h`	`svn log -q`
	1.1	`git show -s --format=%ci`	`svn log`
	2.1	`git show -U100000 --no-renames`	`svn diff -x -U100000 -c`

- `svn log` provides the revision number, author, date, and message; only the number and the date will be extracted to create the commit header information
- `git show -U100000 --no-renames` provides the commit body including all changed artifacts with 100,000 lines of context information for each change to ensure availability of the entire artifact contents; `--no-renames` excludes renaming of artifacts to appear as changes, which is not a typical content change
- `svn diff -x -U100000 -c` provides the commit body similar to the previous Git command, but without explicit exclusion of artifact renaming; SVN does not provide such an option in combination with entire artifact contents

A *commit analyzer* processes the instances of the generic *commit data model* provided by an extractor. In general, it is also an exchangeable plug-in supporting different analysis for the same extracted data. Indeed, we only implemented one specific analyzer, which realizes the commit analysis process as described in Section 4.2. The abstraction of the generic data model allows a common process for both Git- and SVN-based commits. Hence, the identification and classification steps stay the same for each analysis in this paper. However, the matching for relevant artifacts during identification (cf. Section 4.2.1) requires individual regular expressions for each SPL in our analysis. We therefore defined three configuration parameters for the analyzer, which allow the flexible definition of regular expressions for code, build, and variability model artifacts as needed.

The *configuration file* in the lower part of Fig. 6 defines the three regular expressions using the dedicated configuration parameters for the commit analyzer. Further, this file contains a path to a directory for saving the evolution data produced by the analyzer as well as the path to the SPL repository for the extractor. Ultimately, it defines the specific extractor and analyzer to use along other general configuration, like the amount of logging information. The infrastructure reads these parameters to configure ComAnI prior to its actual execution. We provide the main configurations from these files for each evolution analysis of our case studies in the next section.

6. Analysis

In the evolution analysis, we executed our technical realization described in Section 5 in a virtual machine. The exact image used for our analysis along with a description of our host system is publicly available on Kröher et al. (2022b). This repository also contains our suite of implementations and configurations as well as all (intermediate) results. In this section, we first describe the **hardware** and **software** constituting the basic setup used in this paper. We then discuss the **data set** and explain its selection. In particular, we provide some general properties of the SPLs and their repositories to show their diversity. Further, we describe the **execution** of the individual evolution analysis per case study. The **validation** of our results follows at the end of this section.

The **hardware** of the host for our virtual machine consists of a Intel Core i7-8665U CPU with 1.90 GHz and 4 cores. We enabled hyperthreading, which results in 8 logical CPU cores. Further, the host offers 32 GB registered DDR4 RAM (2400 MHz) and a 1TB Samsung SSD 970 EVO Plus. Based on these resources, we configured the virtual machine with access to 2 CPU cores, 16 GB RAM, and 100 GB mass volume storage. In particular, the memory size is crucial to ensure complete processing of extremely large commits. For example, the first commit of the Linux kernel repository introduces the entire kernel version 2.6.12-rc2 as a baseline for further development. However, a reduction of these hardware settings is possible, if such commits do not exist, which then only results in an extension of the execution time.

The **software** is based on Ubuntu 22.04.1 as operating system for the virtual machine. For executing our implementation, we installed OpenJDK 11.0.16, Git 2.34.1, SVN 1.14.1, and R 4.1.2. Further, we extended the basic R environment by packages *Hmisc* and *nortest* for calculating all statistics during analysis.

The **data set** basically consists of four different case studies: Linux kernel (The kernel development community, 2022c), coreboot firmware (coreboot team, 2022), BusyBox UNIX utilities (BusyBox team, 2022), and axTLS embedded SSL (Rich, 2022). While we used the Linux kernel already in our previous work (Kröher et al., 2018), we primarily selected the other projects as they also rely on Kbuild to realize variability. In particular, each of them provides code, build, and variability model artifacts with their individual artifact-specific and variability information as introduced in Section 3.1. This comprehensive data is not available in the required form in other projects we considered, which, for example, mainly use Kconfig for their variability model (Berger, 2012) only or reference variability information across artifact types differently (Gomes et al., 2019). Further, Table 2 shows that the resulting data set already covers a certain diversity ranging from large-scale, complex SPLs (Linux kernel) to smaller projects

Table 2
Properties of case studies and analysis configurations.

			Linux	coreboot	BusyBox	axTLS
Latest project size in lines	*Code*		31,662,545	3,016,636	286,113	32,686
	Code percentage		99.09%	97,52%	97,87%	92,63%
	Variability model		207,070	43,760	2227	782
	VM percentage		0,65%	1,41%	0,76%	2,22%
	Build		82,140	32,979	4000	1820
	Build percentage		0,26%	1,07%	1,37%	5,16%
	Sum (100%)		31,951,755	3,093,375	292,340	35,288
Team size	*Contributors*		13,477	553	244	1
	Forks		44,800	437	491	0
Evolution	*Available commits*	*Count*	1,123,515	49,268	17,554	179
		Initial	`1da177e4c3f4` (2005/04/16)	`77d1a8311f` (2003/04/15)	`cc8ed39b2401` (1999/10/05)	r78 (2007/03/14)
		Latest	`c3e0e1e23c70` (2022/09/28)	`cfec5ddc1605` (2022/09/23)	`c8c1fcdba163` (2022/09/08)	r279 (2019/03/16)
	Analyzed commits	*Count*	1,040,120	49,252	8272	179
		Initial	`686579d95d48` (2005/04/16)	`77d1a8311f` (2003/04/15)	`23b514624` (2002/12/05)	r78 (2007/03/14)
		Latest	`b1cab78ba392` (2022/09/28)	`bbf2706fb4` (2022/09/28)	`6774386d9` (2010/05/09)	r279 (2019/03/16)
Configuration	*Extractor*		Git	Git	Git	SVN
	Analyzer (artifact regex)	*Code*	`*.[chS]`	`*.[chS]`	`*.[chS]`	`*.[chS]`
		Var. model	`Kconfig*`	`Kconfig*`	`Config.in | Config.src`	`Config.in`
		Build	`Makefile* | Kbuild*`	`Makefile* | Kbuild* | *.mk | *.mak`	`Makefile* | Kbuild*`	`Makefile | Rules | .mak*`

(axTLS) and in between (coreboot, BusyBox). This table presents the following information from the respective main branch of the SPL repositories:

- The *latest projects size in lines* shows the individual lines (including comments and empty lines) per artifact type per project at the time of the latest commit available, as well as their share in relation to the sum of all counted lines; we use the same regular expressions for classifying files to artifact types as in the configuration part of the table
- The *team size* by the number of contributors and the number of forks of the main branch as available on Github (Linux kernel, coreboot, BusyBox), or derivable from SourceForge (axTLS)
- The *evolution* consisting of the total number of commits, the initial commit, and the latest one; due to the CEC defined in Section 4.2.1, we differentiate here between all available commits in the repository and those actually analyzed[2]

The **execution** applies our technical realization to the main branch of each case study. We therefore clone the individual repositories and define custom ComAnI configuration files. Besides the aforementioned properties of our data set, Table 2 also includes the main *configuration* parameters for each SPL. It defines the specific extractor and, hence, the commands used for the respective commit extraction (cf. Table 1). Further, Table 2 defines (simplified) regular expressions for the identification of relevant artifacts as described in Section 5. These expressions resulted initially from file types used in related literature (Passos et al., 2016; Tartler et al., 2014; Nadi and Holt, 2011). On this basis, we inspected the available artifacts, the specific file names, and their contents per SPL to update and extend the initial expression to those in the table. Due to the limitation of the BusyBox evolution, we use the partial repository extraction (cf. Section 5) with a custom list of commits. The remaining executions rely on the full repository extraction capability.

The **validation** concerns the correctness of the results from the execution described above. In particular, we need to guarantee that the technical realization with the individual configurations from Table 2 are sufficient for the goal of our analysis. Hence, we tested them against a total number of 84 commits including those from our case studies as well as some artificial ones to inspect special cases individually. We manually counted and categorized all changed lines in these commits and defined the respective numbers as part of automatic test cases. ComAnI and the plug-ins used in this paper pass all of these tests successfully, which allows for a high confidence in the correctness of our results.

7. Results

In this section, we present the results from applying our fine-grained, variability-centric approach to extract and analyze evolution data to the four case studies as described in Section 6. This presentation covers the following perspectives on the individual evolutions of the SPLs:

1. **Changes to general information types** present the absolute numbers and percentages of commits changing artifact-specific information, variability information, both or none of these information types without further differentiation towards affected artifact types.
2. **Changes to artifact types** provide the percentages of commits changing artifact-specific or variability information in code, build, or variability model artifacts.
3. **Line changes per artifact types** extends the commit-based perspective (2) with a line-based perspective on the changes to identify how often and to which extent commits

[2] For BusyBox, we had to further exclude commits before the complete migration to Kbuild and after the introduction of a script for extracting variability model information from code artifacts, which initiated defining variability information of the variability model as part of comments in code artifacts. This mixing of information and artifact types is not supported by our tooling.

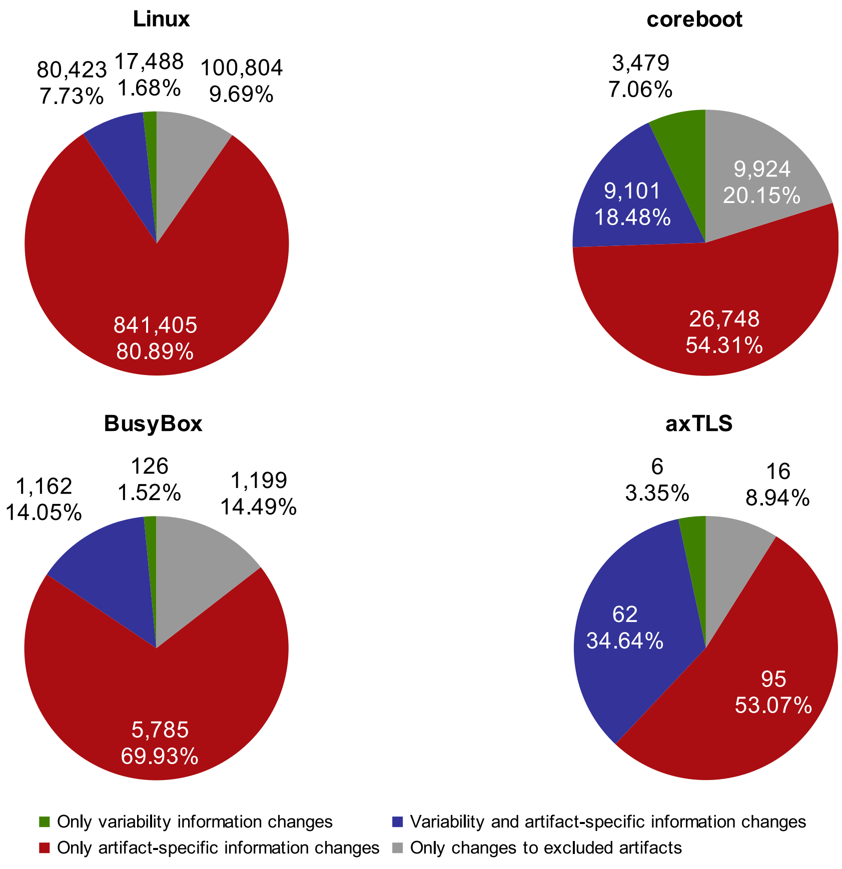


Fig. 7. Commit-based changes to information types.

change a code, build, or variability model lines containing artifact-specific or variability information.

The three perspectives continuously lower the level of abstraction of our results. While perspective 1 hides details to present the individual evolutions on the level of the fundamental information types of a SPL, perspective 2 refines these results towards an artifact-based level. Perspective 3 constitutes the finest level of abstraction as it also considers the content of each type of artifact.

7.1. Changes to general information types

The overview on the commits and their impact on artifact-specific and variability information in general is presented in Fig. 7. It provides a basic understanding of which type of information is changed more frequently regardless of the type of artifact the information belongs to. Hence, the results of this section apply to the fundamental information types of our case studies.

In the **Linux** kernel evolution, the largest category of commits with about 81% exclusively changes artifact-specific information. The second largest category of commits with about 10% changes excluded artifacts exclusively. These artifacts are not in the focus of this paper, like documentation or device tree source files. Commits changing both artifact-specific and variability information constitute the third largest category of commits with about 8%. The smallest category of commits with about 2% consists of commits exclusively changing variability information. An interesting insight of this analysis is that only 9% of the analyzed commits contain changes that affect the variability information of any artifact in the Linux kernel.

The **coreboot** evolution shows a similar order regarding the sizes of the commit categories as for Linux. However, a shift from commits changing artifact-specific information only to the other categories exist. This former category still presents the largest one with about 54%, while commits changing excluded artifacts exclusively (about 20%) and commits changing both information types (about 19%) are doubling. The commits changing variability information exclusively remain the smallest category with about 7%. In total, about a quarter of all commits change variability information regardless of the type of artifacts in the evolution of coreboot.

The categorization of the **BusyBox** commits also provides the same order in size as for Linux and coreboot. Further, a similar shift from exclusive artifact-specific information changes to the other categories as for coreboot exists. However, the absolute distances of the percentages per category reveal that the BusyBox evolution resembles Linux more than coreboot. The total number of commits changing variability information increases nevertheless from only 9% for Linux to about 16% for Busybox.

The evolution of **axTLS** mostly concerns artifact-specific information (53% of all commits) as for the other SPLs. The main difference is the comparably high number of commits changing artifact-specific and variability information together. This category constitutes the second largest one of commits with about 35% and, hence, exceeds the number of commits changing excluded artifacts exclusively (about 9%). This is a unique property of the axTLS evolution, which also results in the highest total number of commits changing variability information with 38%. Another interesting insight of this analysis is that axTLS is closer to the evolution of coreboot regarding exclusive artifact-specific information changes, while it is closer to Linux in terms of exclusive changes to variability information and as well as excluded artifacts.

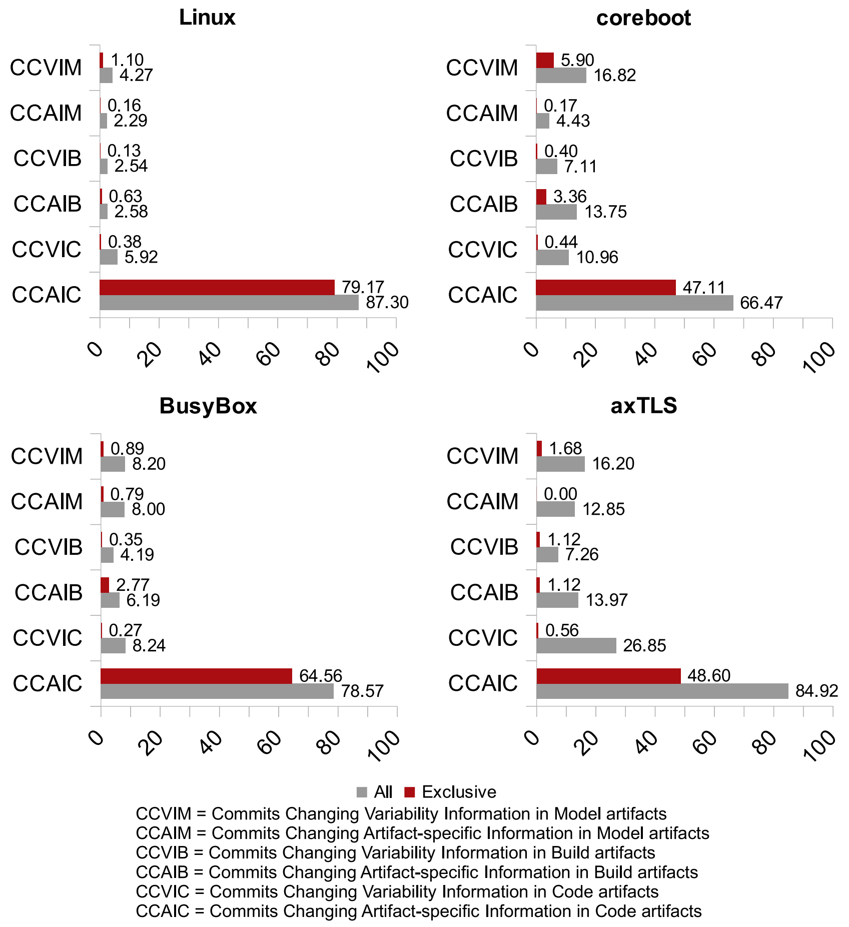


Fig. 8. Commit-based changes to artifact information types (in %). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

7.2. Changes to artifact types

The statistics in the previous section abstract from the impact the commits have on the different types of artifacts as well as the changes to their artifact-specific and variability information. In order to understand the individual evolution of artifacts and information in the focus of this paper, we lower the level of abstraction towards an artifact-based one. We therefore refine the commit categories presented in Fig. 7 in this section. In particular, we are interested in the numbers of commits changing artifact-specific or variability information in code, build, and variability model artifacts. Fig. 8 presents the resulting refinement. It illustrates the percentages of all analyzed commits per case study (cf. Table 2) along the following commit categories for each type of artifacts:

- **Commits Changing Artifact-specific Information** change at least one line containing this type of information in the respective type of artifact. Further, we differentiate between *all* commits introducing such a change (gray bars in Fig. 8) and those applying *exclusive* changes of this type (red bars in Fig. 8). The former commits may also introduce changes to variability information or other types of artifacts. The latter commits neither introduce changes to variability information nor to other types of artifacts (subset of the former).
- **Commits Changing Variability Information** change at least one line containing variability information in the respective type of artifact. The additional differentiation between *all* commits and those with *exclusive* changes of this type also applies here as described above, but for variability information.

The largest category of commits in the **Linux** kernel evolution changes artifact-specific information in code artifacts (*CCAIC*). In about 91% of all of these cases, artifact-specific information is changed exclusively. Variability information in code artifacts (*CCVIC*) changes second most frequently, but significantly less (about 6% of all commits, about 0,4% exclusively). Commits changing the same information type in variability model artifacts (*CCVIM*) represent the third largest category with about 4% of all analyzed commits out of which about 26% change this type of information exclusively. These exclusive changes occur even more often than exclusive changes to this type of information in code artifacts (*CCVIC*), which is to be expected as the variability model artifacts mostly contain variability information. However, there also exists a small number of commits (about 0.2%), which exclusively change artifact-specific information, like help texts, in variability model artifacts (*CCAIM*). In general, *CCVIM* occur more frequently than such changes in build artifacts *CCVIB*, but less than in code artifacts (*CCVIC*) regarding all commits. The numbers of commits changing information in build artifacts (*CCAIB* and *CCVIB*) are almost equal. In particular, the numbers of all commits affecting build artifacts are comparable with the number of commits changing artifact-specific information in variability model artifacts (*CCAIM*).

In the **coreboot** evolution, commits changing artifact-specific information in code artifacts (*CCAIC*) are not as frequent as in

Linux, but still constitute the dominant category of commits. Further, about 71% of all of these cases change this type of information in this type of artifacts exclusively, which is another reduction with respect to Linux. The second largest category of commits with about 17% changes variability information in variability model artifacts (*CCVIM*), followed by changes to artifact-specific information in build artifacts (*CCAIB*) and variability information changes in code artifacts (*CCVIC*). This order of categories differs significantly from the previous case study. The coreboot commits apply more changes to variability model and build artifacts in general as well as to variability information in particular. The latter increase continuous the shift from pure artifact-specific information changes to changes affecting both information types simultaneously identified in Section 7.1. However, exclusive changes to variability information do not increase likewise for all categories, like *CCVIC*. The changes to variability information in build artifacts (*CCVIB*) and to artifact-specific information in variability model artifacts (*CCAIM*) represent the smallest categories as in Linux.

The distribution of the **BusyBox** commits reveals an equal order of the three largest categories as in Linux and, hence, respective differences to coreboot. Changes to artifact-specific information in code artifacts (*CCAIC*) constitute the largest category with about 79%, while variability information changes in the same artifacts (*CCVIC*) and in variability model artifacts (*CCVIM*) follow almost equally with about 8%. However, the quantity of commits changing artifact-specific information in variability model artifacts (*CCAIM*) is also about 8% and only slightly less. The smallest categories concern changes to build artifacts. The respective commits change artifact-specific information (*CCAIB*) with about 6% more frequently than variability information (*CCVIB*) with about 4%. Hence, the BusyBox evolution is the only one with a distinct order along the types of artifacts: most changes apply to code artifacts, followed by variability model and build artifact. Further, significant findings, e.g. regarding exclusive changes, do not exist. They appear slightly more often, but still comparable to those in Linux regarding their ratio to all commits.

The evolution of **axTLS** also shares the same order of the three largest commit categories with Linux (*CCAIC*, *CCVIC*, and *CCVIM*) and even the fourth largest category (*CCAIB*) with BusyBox additionally. Further, the axTLS commits apply changes to build and variability model artifacts in a similar frequency than those of the coreboot evolution in general. An unique characteristic of axTLS is the ratio of commits changing a specific type of information exclusively. For example, only about 57% of all commits changing artifact specific information in code artifacts (*CCAIC*) introduce these such changes without changing other information or artifacts. While this is the lowest value for this ratio in our entire analysis, it also represents the ratio of exclusive changes for the other information and artifacts types in this SPL. In the most extreme case of commits changing artifact-specific information in variability model artifacts *CCAIM*, not a single exclusive change of this type exists. These values indicate rather heterogeneous changes concerning the affected artifacts and contents in comparison with the other case studies.

7.3. Line changes per artifact types

In this section, we further detail our results to identify how many commits change a certain number of code, build, or variability model lines containing artifact-specific or variability information. This line-based level of abstraction reveals how often and to which extent the content of these artifacts is changed in general and, in particular, the variability information. We therefore define a set of intervals of changed lines to classify the intensity of those changes. We then use these intervals to classify the commits accordingly as presented in Fig. 9.[3] It visualizes the percentage relative to the total number of all analyzed commits for each interval (*I1*-*I5*) and category (*CMLVI* to *CCLAI*). For example, interval *I1* contains all commits, which do not change a single line in the respective category. Hence, about 13% of all analyzed Linux commits belong to this interval for changed code lines containing artifact-specific information (*CCLAI*). Due to this categorization, commits may be counted multiple times: a commit, which only changes five lines of the general program definition in code artifacts, belongs to interval *I2* for category *CCLAI*, but also to interval *I1* for all other categories.

In the **Linux** kernel evolution, the types of artifacts in the focus of this paper are changed rather infrequently, except for changed code lines containing artifact-specific information (*CCLAI*). Around 96–98% of the commits do not change a single line in build and variability model artifacts. Even if such changes occur, they are rather small: only 2–4% of the commits change 1–10 lines, while more changes occur in around 0.1–0.4% of all commits or less. In build artifacts, there is no significant difference between the frequency of changes to artifact-specific (*CBLAI*) and variability information (*CBLVI*). In variability model artifacts, however, smaller changes (interval *I2*) to variability information (*CMLVI*) occur more often than to artifact-specific information (*CMLAI*). In contrast, the frequency of changes to code artifacts differs with respect to the information types. Changes to code lines containing variability information (*CCLVI*) are classified rather similar to those in variability model artifacts. Here, around 93% of the commits do not change a single line and only some commits change 1–10 lines. Code lines containing artifact-specific information (*CCLAI*), however, are subject to larger and more frequent changes. For *CCLAI*, only about 13% of the commits do not introduce changes, while around 41% and 29% of the commits change 1–10 lines and 11–50 lines of the general program definition, respectively. Further, even larger changes (intervals *I4* and *I5*) to artifact-specific lines of code occur more often as the small changes to build and variability model artifacts.

The **coreboot** commits change the content of build, and variability model artifacts slightly more frequently than the commits in Linux. In about 4–13% of the commits, 1–10 lines change in these types of artifacts in general. Further, changes apply more often to artifact-specific information (11% for *CBLAI*) than to variability information (about 7% for *CBLVI*) in build artifacts, while vice-versa for variability model artifacts. In particular, coreboot provides the second highest number of commits introducing small changes (interval *I2*) and the highest numbers for larger changes (intervals *I3* to *I5*) to variability information in variability model artifacts (*CMLVI*). In contrast, changes to artifact-specific information in code artifacts (*CCLAI*) are comparably infrequent. About 34% of the commits do not change a single line. Larger changes (intervals *I3* to *I5*) also occur less often than in the other case studies, but still remain the predominant category. Variability information in code artifacts (*CCLVI*) changes by 1–10 lines in about 9% of all analyzed commits. This is again the second highest number in our analysis for this category and interval. Even for the larger intervals, coreboot has the second highest number for *CCLVI*.

The commits of the **BusyBox** evolution introduce changes to the majority of categories similar to the commits in Linux. In particular, around 92–96% of the commits neither change any build or variability model content in general, nor any variability

[3] The complete definition contains 12 different intervals, which starts with those presented in Fig. 9 as the majority of changes affects 0–200 lines. We continuously enlarge the remaining intervals to cover the largest changes as well. However, we refrain from presenting these additional ones as this would significantly increase the complexity of the individual charts, while adding less than 3% of commits of the respective SPL.

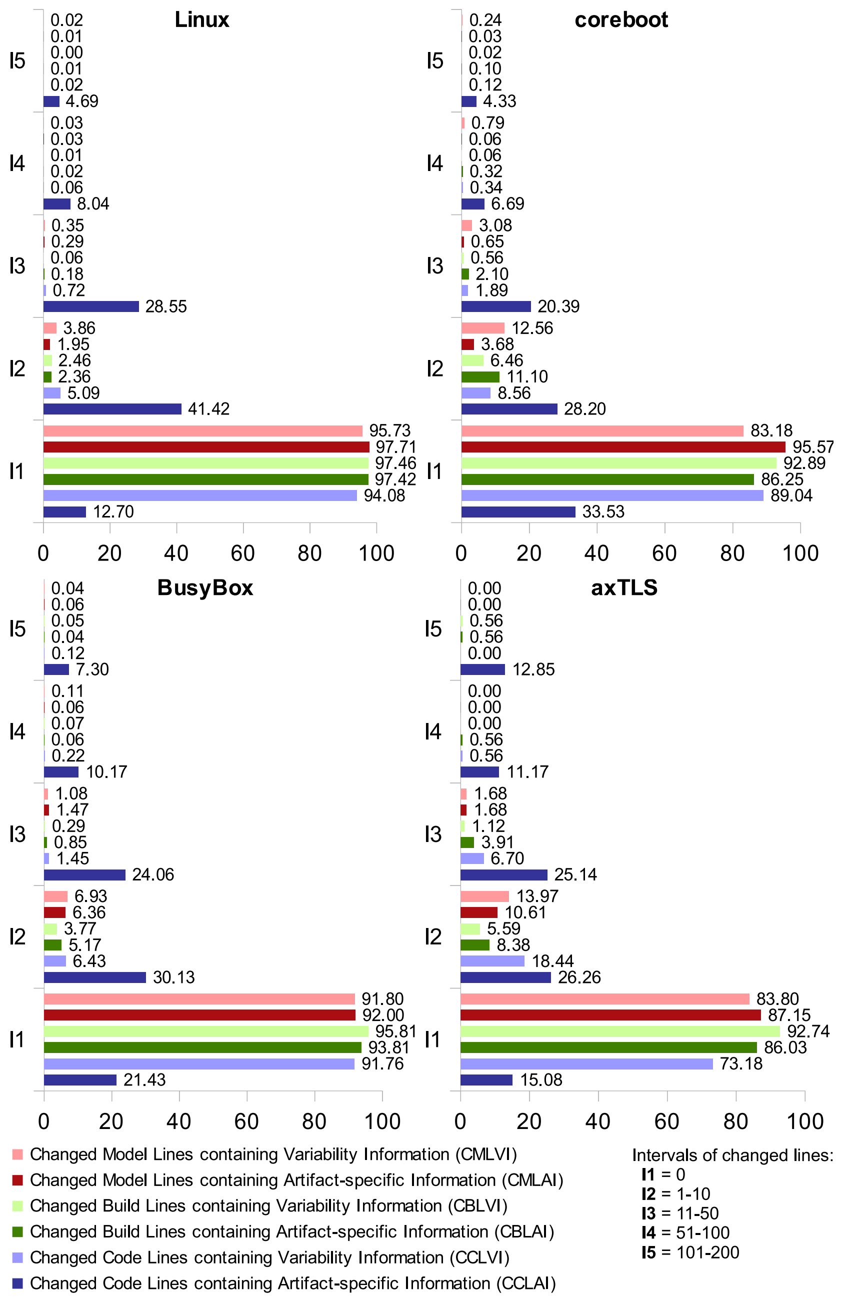


Fig. 9. Frequency of line changes (in %).

information in code artifacts. In contrast, commits changing 1–10 lines of artifact-specific information in code artifacts (*CCLAI*) occur less frequent than in Linux, but more often than in coreboot (about 21% for intervals *I1*). Further, a higher number of larger changes (intervals *I4* and *I5*) to *CCLAI* exist in BusyBox than in the previous case studies. Commits regarding variability model artifacts likewise concern both types of information ranging from about 6–7% changing 1–10 lines to 0.04–0.06% changing 101–200 lines. For build artifacts, changes apply slightly more frequently to artifact specific information than to variability information independent of the interval. While this difference is similar to the distribution of build information changes in the coreboot evolution, it is not as distinct as in that SPL.

In the **axTLS** evolution, interval *I1* also represents the predominant one with respect to changes to build and variability model artifacts in general, and variability information in code artifacts (*CCLVI*). However, with about 73–93% the actual numbers for these categories are smaller than in the previous case studies. This reduction in interval *I1* indicates more changes to the artifact and information types, which the respective numbers for the other intervals support. For example, axTLS provides the highest numbers for CCLVI with about 18% in interval *I2*, about 7% in interval textitI3, and even 0.6% in interval textitI4. Further, changes to artifact-specific information in code artifacts (*CCLAI*) affect 1–10 lines (interval *I2*) almost as often as 11–50 lines (interval *I3*). In particular, the about 13% of commits changing 101–200 lines in category *CCLAI* are again the highest number in our analysis. Variability model changes in interval *I2* also occur most frequently in axTLS and remain their comparably high numbers for larger changes regardless of the affected information type. In contrast, changes to build artifacts range between the respective numbers of BusyBox and coreboot. For this type of artifact, only about 6% of all analyzed commits introduce smaller changes to variability information (interval *I2*), while larger changes are introduced by only 1% or less.

8. Discussion

In this section, we discuss the results presented in Section 7 in relation to the research questions raised in Section 1. Sections 8.1 to 8.3 structure this discussion along RQ1 to RQ3. The answers to these questions provide fine-grained and comparative insights in the intensities of variability changes in different types of artifacts for different SPLs. Further, we briefly discuss in Section 8.4 the implications of this comparison on supporting SPL evolution in general, and, in particular, on the utilization we proposed previously (cf. Section 3.2).

8.1. Changes to variability information

In our previous analysis of the Linux kernel evolution (Kröher et al., 2018), we found that about 80% of all the analyzed 662.110 commits affect only artifact-specific information, despite the Linux kernel being a highly-configurable software system. We confirmed this finding in our current study, which includes the previously analyzed commits and roughly doubled their amount to 1,123,515 analyzed commits in summary. However, a comparison with the other three projects (coreboot, BusyBox, axTLS) also reveals that the variability information change-intensity of the Linux kernel is higher than the median percentage of these changes for all four projects changing only artifact-specific content (62%). The lowest amount of artifact-specific changes has axTLS with 53%, which is still the majority of all changes. Changes to variability information occur in median at about 16% of all commits over all four projects, and only around 2.5% of commits change this information exclusively. Interestingly, by far the highest amount of commits changing variability and artifact-specific information is from the axTLS project with 34%, while the most commits exclusively changing variability information are from coreboot with only 7.1%.

In conclusion and as answer to RQ1, the comparison of different SPLs shows a certain variety in the amount of changes to variability information, with a higher standard deviation for artifact-specific (11%) and combined variability and artifact-specific changes (9.9%), and a lower standard deviation for variability information changes (2.2%). Overall, artifact-specific changes are the most frequent in all four projects, while variability related changes are the minority. This supports our insights based on the single analysis of the Linux kernel, that for the more stable phases of SPL development the variability information does not change significantly with respect to the overall number of changes.

8.2. Variability changes to artifact types

In median, most of the changes to variability information over all four SPLs affect variability model artifacts (12%), followed by code artifacts (9.6%), and build artifacts (5.7%), as presented in Section 7.2. This is different to our previous study of the Linux kernel (Kröher et al., 2018), where most changes to variability information happen in code artifacts. However, the individual picture of the analyzed SPLs is not as clear as the median values suggest: For Linux and axTLS, most of the variability changes also happen in code files (5.9% and 27% respectively), and for BusyBox the changes to variability information are nearly equally distributed between code (8.24%) and variability model artifacts (8.20%). Only in coreboot the changes to variability information actually occur more often in variability model artifacts (17%) than in code artifacts (11%). These differences can be explained with the high variety (standard deviation of 8.2% for code artifacts and 5.3% for variability model artifacts) of the results for each SPL.

As a consequence, the answer to RQ2 is that the intensity of variability information in code and variability model artifacts vary to a higher degree of about five to eight percent between different SPLs. Interestingly, this is also true for artifact-specific changes to code (standard deviation of 8.1%) and build artifacts (standard deviation of 4.9%), and to a lower degree for artifact-specific changes to variability model artifacts (standard deviation of 4.0%). Changes to variability information in build artifacts are the most uncommon (median of 5.7%) and have the lowest standard deviation (2%) over all four SPLs.

8.3. Average line changes affecting variability

The frequency of line changes containing variability information in the four SPLs ranges in the same order of magnitude for all three artifact types, as described in Section 7.3. This is aligned with the Linux kernel evolution results from our previous study (Kröher et al., 2018). Most commits that change variability information are only 1–10 lines small, where commits changing variability information in variability model artifacts take the biggest share with 9.8% in median (with a standard deviation of 4.1%) over all four projects, followed by a median of 7.5% (with a standard deviation of 5.2%) for 1–10 line changes in code artifacts and a median of 4.7% (with a standard deviation of 1.6%) for 1–10 line changes in build artifacts. However, similar to the results per artifact type in the previous section, the individual results per SPL vary: axTLS has the largest percentage of commits bigger than 10 lines that change variability information in code (8.4% of all commits from axTLS) or build artifacts (1.7% of all commits from axTLS), while coreboot has the largest percentage of commits bigger than 10 lines that change variability information in variability model artifacts (4.3% of all commits from coreboot). Nevertheless, commits in the range of 1–10 lines changing variability information are always more frequent than bigger commits changing variability information for each SPL.

The answer to the final research question RQ3 is therefore that commits affecting variability information concern only 1–10 lines of code for all artifact types in all SPLs on average. Surprisingly, the smallest project in terms of lines of code (axTLS) tends to have the biggest commits changing variability information for at least two of the affected artifact types (code and build). On the other hand, the Linux kernel as biggest project has the smallest commits changing variability information over all artifact types.

8.4. Implications and utilization

Our fine-grained, variability-centric analysis of the evolution of four different SPLs confirms that developers change comparatively little variability information over time. These findings assure us that considering change information is a good opportunity to improve approaches and tools for supporting SPL evolution. For example, our findings can be used to reduce analysis time for variability-sensitive analyses, as they do not need to be triggered after every commit. However, exploiting our results in this way is limited to block-based analyses, like the variable (un)dead code block analysis (Tartler et al., 2011) or configuration mismatches (El-Sharkawy et al., 2017). Analyses relying on detailed code semantics, like type-checking (Kästner et al., 2011) or static analysis (Liebig et al., 2013), are out of scope.

The performance analysis based on the exclusive consideration of the Linux kernel (Kröher et al., 2022a) already demonstrated a significant analysis time reduction (from 17 min to 3 s in median) as described in Section 3.2. However, the variability information changes in the Linux kernel evolution are the least frequent in comparison to the other SPLs in this paper. We unsuccessfully searched for reasons by comparing their coding and commit guidelines, lines of code, number of maintainers, purpose, and age. In particular, the differences between coreboot and Linux are surprising, as both are very similar with regard to the aforementioned criteria and in contrast to the other two projects. Hence, we cannot report clear reasons for the differences, but on their impact on evolution support as in Kröher et al. (2022a). In general, we expect a slightly less distinct performance gain as with Linux, while there still will be significant improvements per SPL. Our previous opportunities for improvement (Kröher et al., 2018) still hold even in the light of a diverse range of SPLs.

9. Threats to validity

The analysis in this paper only considers (a subset of) the evolution data in the public repositories, which is a threat to *internal validity*. It does not cover the entire lifetime of any of the SPLs and, in particular, the beginning of their developments. An extension of our analysis to this missing data may change our results. For example, we assume different intensities of changes in the early development phases of a SPL in general. However, the presented results are valid for the more stable phases after setting up a SPL. Further, different coding-conventions will lead to different results as we count changes line-by-line. For instance, multiple statements in a single changed line are counted as a single change to either artifact-specific or variability information and not independently. This is an open threat to *internal validity*, which requires substantial changes to the analysis algorithms presented in this paper.

The first threat to *construct validity* arises from our mappings between specific files types of the respective SPL and the general types of artifacts in the focus of our analysis (cf. Section 6). These mappings are based on those being used in related literature (Passos et al., 2016; Tartler et al., 2014; Nadi and Holt, 2011) as well as our own inspection of the individual SPLs. However, other mappings exist, like the one by Dintzner et al. (2016), which include additional file types for some of our artifact types. For the Linux kernel, we already showed that the amount of these additional files is relatively small compared to the overall code base (Kröher et al., 2018). Further, our mappings correspond to those typically used in related analysis as indicated above. Hence, we argue that we use well-accepted mappings and additional inclusions of other files will not fundamentally change our results.

The second threat to *construct validity* concerns the patterns and expressions we used to identify and classify changed lines as containing variability or artifact-specific information in the respective types of artifacts (cf. Section 4.2). We therefore validated our realization using 84 commits as described in Section 6. However, as the sample commit size is quite small, there is a risk of false-positive or missing detections by these patterns and expressions.

The third threat to *construct validity* concerns the intervals classifying the changes in Section 7.3. The definition of these intervals is based on a subjective valuation of the most meaningful classifications by the authors. While we could have used other intervals, this would not change the general findings nor the conclusions.

The selection of our four case studies and the specific focus of our analysis threaten *external validity*. All SPLs use annotative variability realization techniques in code and build artifacts and the Kconfig language to define the variability model. Hence, we cannot generalize the results to other SPLs, in particular, those using compositional variability realization techniques or other approaches to model variability. For other SPLs using the same techniques, the results may be different, which the already identified differences in this paper indicate. However, our case studies represent a rather diverse set of SPLs regarding their project and development size as well as their evolution. This diversity fosters our main findings with respect to their generality at least to some extent.

Finally, the results and discussions presented in this paper are limited to block-based analyses, which is a second threat to *external validity*. They can only partially be transferred to other types of analyses, like type-checking. These types of approaches require a different design of the evolution analysis, which also considers additional information, like the data- or control-flow.

A thorough discussion on threats to validity also needs to address *conclusion validity*. However, our results are mostly descriptive in nature and do not include any advanced conclusions drawn from our analysis, besides the discussion of their implications and utilization. The core result, that developers change comparatively little variability information across all the different artifact types, directly stems from our fine-grained analysis approach for which we discussed relevant threats above. Hence, we cannot report any further threats to conclusion validity.

10. Conclusion

We presented a significant extension of our previous work on understanding the evolution of Software Product Lines (SPLs) (Kröher et al., 2018). It consists of an application of our fine-grained, variability-centric analysis approach to four different SPLs: Linux kernel, coreboot firmware, BusyBox UNIX utilities, and axTLS embedded SSL. The approach is based on the differentiation between artifact-specific and variability information in code, build, and variability model artifacts to identify the intensity (the frequency and the amount) with which developers change variability information in practice. The discussed extraction and analysis processes consider these changes independent from their relation to a specific feature and, hence, complement existing work. We applied this approach to the commit histories of each SPL, presented the results of the individual analyses, and compared them for a broader understanding of SPL evolution. Further, we reviewed our previous discussions (Kröher et al., 2018) and follow-up results (Kröher et al., 2022a) in the light of the additional evolution data.

The extension of our analysis confirms our initial main results. In general, changes to variability information occur infrequently and only affect small parts of the analyzed artifact types. However, differences between the analyzed SPLs exist. In particular, the frequency of variability changes varies significantly. While some extreme cases leading to this variation also reduce our initial expected benefit from exploiting our findings, the average gathered evolution data confirms the optimization potential for certain evolutionary analysis and verification tasks. In this regard, our extended findings may encourage the research community further to consider fine-grained information about the actual changes to variability information in the design of new approaches for SPL evolution support.

In future work, we envision two research directions based on the results in this paper. On the one hand, an application of our prototype for incremental SPL verification to the evolution of coreboot, BusyBox, and axTLS can be worthwhile to foster our previous performance results (Kröher et al., 2022a). On the other hand, an analysis of the evolution of SPLs employing other techniques and technologies to realize variability would complement our current research. For example, contrasting the intensity of variability changes between annotative and compositional SPLs (Kästner et al., 2008) represents a reasonable extension to our current evolution knowledge gained from the presented analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The research data and code is publicly available; the manuscript includes the respective links

References

Adams, B., Schutter, K., Tromp, H., Meuter, W., 2007. The evolution of the linux build system. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. 8, 1–16. http://dx.doi.org/10.14279/tuj.eceasst.8.115.

Berger, T., 2012. Variability modeling in the wild. In: 16th International Software Product Line Conference, Vol. 2. ACM, New York, NY, USA, pp. 233–241. http://dx.doi.org/10.1145/2364412.2364452.

Borba, P., Teixeira, L., Gheyi, R., 2012. A theory of software product line refinement. Theoret. Comput. Sci. 455, 2–30. http://dx.doi.org/10.1007/978-3-642-14808-8_2.

BusyBox team, 2022. BusyBox UNIX utilities. https://github.com/mirror/busybox, Accessed 2022/10/17; cloned 2022/09/29, 9:39am using https://github.com/mirror/busybox.git.

Chacon, S., Straub, B., 2014. Pro Git. A Press, New York, NY, USA.

Chaikalis, T., Chatzigeorgiou, A., Examiliotou, G., 2015. Investigating the effect of evolution and refactorings on feature scattering. Softw. Qual. J. 23 (1), 79–105. http://dx.doi.org/10.1007/s11219-013-9204-4.

coreboot team, 2022. coreboot firmware. https://github.com/coreboot/coreboot, Accessed 2022/10/17; cloned 2022/09/29, 9:15am using https://github.com/coreboot/coreboot.git.

Dietrich, C., Tartler, R., Schröder-Preikschat, W., Lohmann, D., 2012. A robust approach for variability extraction from the linux build system. In: 16th International Software Product Line Conference, Vol. 1. ACM, New York, NY, USA, pp. 21–30. http://dx.doi.org/10.1145/2362536.2362544.

Dintzner, N., Deursen, A., Pinzger, M., 2016. FEVER: Extracting feature-oriented changes from commits. In: 13th International Conference on Mining Software Repositories. ACM, New York, NY, USA, pp. 85–96. http://dx.doi.org/10.1145/2901739.2901755.

Dintzner, N., Deursen, A., Pinzger, M., 2017. Analysing the linux kernel feature model changes using FMDiff. Softw. Syst. Model. 16 (1), 55–76. http://dx.doi.org/10.1007/s10270-015-0472-2.

El-Sharkawy, S., Krafczyk, A., Schmid, K., 2017. An empirical study of configuration mismatches in linux. In: 21st International Systems and Software Product Line Conference, Vol. A. ACM, New York, NY, USA, pp. 19–28. http://dx.doi.org/10.1145/3106195.3106208.

Godfrey, M., Tu, Q., 2000. Evolution in open source software: A case study. In: International Conference on Software Maintenance. IEEE Computer Society, Washington, DC, USA, pp. 131–142. http://dx.doi.org/10.1109/ICSM.2000.883030.

Gomes, K., Teixeira, L., Alves, T., Ribeiro, M., Gheyi, R., 2019. Characterizing safe and partially safe evolution scenarios in product lines: An empirical study. In: 13th International Workshop on Variability Modelling of Software-Intensive Systems. ACM, New York, NY, USA, pp. 1–9. http://dx.doi.org/10.1145/3302333.3302346.

Hellebrand, R., Silva, A., Becker, M., Zhang, B., Sierszecki, K., Savolainen, J., 2014. Coevolution of variability models and code: An industrial case study. In: 18th International Software Product Line Conference, Vol. 1. ACM, New York, NY, USA, pp. 274–283. http://dx.doi.org/10.1145/2648511.2648542.

Holdschick, H., 2012. Challenges in the evolution of model-based software product lines in the automotive domain. In: 4th International Workshop on Feature-Oriented Software Development. ACM, New York, NY, USA, pp. 70–73. http://dx.doi.org/10.1145/2377816.2377826.

Hunsen, C., Zhang, B., Siegmund, J., Kästner, C., Leßenich, O., Becker, M., Apel, S., 2016. Preprocessor-based variability in open-source and industrial software systems: An empirical study. Empir. Softw. Eng. 21 (2), 449–482. http://dx.doi.org/10.1007/s10664-015-9360-1.

Israeli, A., Feitelson, D., 2010. The linux kernel as a case study in software evolution. J. Syst. Softw. 83 (3), 485–501. http://dx.doi.org/10.1016/j.jss.2009.09.042.

Kästner, C., Apel, S., Kuhlemann, M., 2008. Granularity in software product lines. In: 30th International Conference on Software Engineering. ACM, New York, NY, USA, pp. 311–320. http://dx.doi.org/10.1145/1368088.1368131.

Kästner, C., G. Giarrusso, P., Rendel, T., Erdweg, S., Ostermann, K., Berger, T., 2011. Variability-aware parsing in the presence of lexical macros and conditional compilation. In: 2011 ACM International Conference on Object-Oriented Programming Systems Languages and Applications. ACM, New York, NY, USA, pp. 805–824.

Kröher, C., 2022. Commit Analysis infrastructure (ComAnI). https://github.com/CommitAnalysisInfrastructure, Accessed 2022/10/17.

Kröher, C., Flöter, M., Gerling, L., Schmid, K., 2022a. Incremental software product line verification - A performance analysis with dead variable code. Empir. Softw. Eng. 27 (68), 1–41. http://dx.doi.org/10.1007/s10664-021-10090-6.

Kröher, C., Gerling, L., Schmid, K., 2018. Identifying the intensity of variability changes in software product line evolution. In: 22nd International Systems and Software Product Line Conference, Vol. 1. ACM, New York, NY, USA, pp. 54–64. http://dx.doi.org/10.1145/3233027.3233032.

Kröher, C., Gerling, L., Schmid, K., 2022b. Comparing the intensity of variability changes in software product line evolution - related research artifacts. https://doi.org/10.5281/zenodo.7273340, Accessed 2022/11/02.

Kröher, C., Schmid, K., 2017a. A Commit-Based Analysis of Software Product Line Evolution: Two Case Studies. Technical Report SSE 2/17/E, University of Hildesheim.

Kröher, C., Schmid, K., 2017b. Towards a better understanding of software product line evolution. In: Softwaretechnik-Trends, Vol. 37:2. Gesellschaft für Informatik e.V., Fachgruppe PARS, Berlin, Germany, pp. 40–41.

Liebig, J., von Rhein, A., Kästner, C., Apel, S., Dörre, J., Lengauer, C., 2013. Scalable analysis of variable software. In: 9th Joint Meeting on Foundations of Software Engineering. ACM, New York, NY, USA, pp. 81–91.

Lotufo, R., She, S., Berger, T., Czarnecki, K., Wasowski, A., 2010. Evolution of the linux kernel variability model. In: 14th International Conference on Software Product Lines: Going beyond. Springer-Verlag, Berlin, Heidelberg, Germany, pp. 136–150. http://dx.doi.org/10.1007/978-3-642-15579-6_10.

Michelon, G.K., Assunção, W.K.G., Obermann, D., Linsbauer, L., Grünbacher, P., Egyed, A., 2021. The life cycle of features in highly-configurable software systems evolving in space and time. In: 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. ACM, New York, NY, USA, pp. 2–15. http://dx.doi.org/10.1145/3486609.3487195.

Nadi, S., Berger, T., Kästner, C., Czarnecki, K., 2015. Where do configuration constraints stem from? An extraction approach and an empirical study. IEEE Trans. Softw. Eng. 41, 820–841. http://dx.doi.org/10.1109/TSE.2015.2415793.

Nadi, S., Holt, R., 2011. Make it or break it: Mining anomalies from linux kbuild. In: 18th Working Conference on Reverse Engineering. IEEE, Los Alamitos, CA, USA, pp. 315–324. http://dx.doi.org/10.1109/WCRE.2011.46.

Nadi, S., Holt, R., 2012. Mining Kbuild to detect variability anomalies in linux. In: 16th European Conference on Software Maintenance and Reengineering. IEEE Computer Society, Washington, DC, USA, pp. 107–116. http://dx.doi.org/10.1109/CSMR.2012.21.

Nadi, S., Holt, R., 2014. The linux kernel: A case study of build system variability. J. Softw.: Evol. Process. 26 (8), 730–746. http://dx.doi.org/10.1002/smr.1595.

Neves, L., Borba, P., Alves, V., Turnes, L., Teixeira, L., Sena, D., Kulesza, U., 2015. Safe evolution templates for software product lines. J. Syst. Softw. 106 (C), 42–58. http://dx.doi.org/10.1016/j.jss.2015.04.024.

Neves, L., Teixeira, L., Sena, D., Alves, V., Kulesza, U., Borba, P., 2011. Investigating the safe evolution of software product lines. In: 10th ACM International Conference on Generative Programming and Component Engineering. ACM, New York, NY, USA, pp. 33–42. http://dx.doi.org/10.1145/2189751.2047869.

Oliveira, R., Cafeo, B., Hora, A., 2019. On the evolution of feature dependencies: An exploratory study of preprocessor-based systems. In: 13th International Workshop on Variability Modelling of Software-Intensive Systems. ACM, New York, NY, USA, pp. 1–9. http://dx.doi.org/10.1145/3302333.3302342.

Passos, L., Czarnecki, K., Wasowski, A., 2012. Towards a catalog of variability evolution patterns: The linux kernel case. In: 4th International Workshop on Feature-Oriented Software Development. ACM, New York, NY, USA, pp. 62–69. http://dx.doi.org/10.1145/2377816.2377825.

Passos, L., Guo, J., Teixeira, L., Czarnecki, K., Wasowski, A., Borba, P., 2013. Coevolution of variability models and related artifacts: A case study from the linux kernel. In: 17th International Software Product Line Conference. ACM, New York, NY, USA, pp. 91–100. http://dx.doi.org/10.1145/2491627.2491628.

Passos, L., Teixeira, L., Dintzner, N., Apel, S., Wasowski, A., Czarnecki, K., Borba, P., Guo, J., 2016. Coevolution of variability models and related software artifacts – A fresh look at evolution patterns in the linux kernel. Empir. Softw. Eng. 21 (4), 1744–1793. http://dx.doi.org/10.1007/s10664-015-9364-x.

Rich, C., 2022. axTLS embedded SSL. https://sourceforge.net/projects/axtls/, Accessed 2022/10/17; cloned 2022/09/29, 9:24am using https://svn.code.sf.net/p/axtls/code/trunk.

Sampaio, G., Borba, P., Teixeira, L., 2016. Partially safe evolution of software product lines. In: 20th International Systems and Software Product Line Conference. ACM, New York, NY, USA, pp. 124–133. http://dx.doi.org/10.1145/2934466.2934482.

Schmid, K., El-Sharkawy, S., Kröher, C., 2019. Improving software engineering research through experimentation workbenches. In: From Software Engineering to Formal Methods and Tools, and Back. Springer-Verlag, Berlin, Heidelberg, pp. 67–82. http://dx.doi.org/10.1007/978-3-030-30985-5_6.

Svahnberg, M., Bosch, B., 1999. Evolution in software product lines: Two cases. J. Softw. Mainten.: Res. Pract. 11 (6), 391–422. http://dx.doi.org/10.1002/(SICI)1096-908X(199911/12)11:6\T1\textless391::AID-SMR199\T1\textgreater3.0.CO;2-8.

Tartler, R., Dietrich, C., Sincero, J., Schröder-Preikschat, W., Lohmann, D., 2014. Static analysis of variability in system software: The 90,000 #ifdefs issue. In: USENIX Annual Technical Conference. USINEX, Berkeley, CA, USA, pp. 421–432.

Tartler, R., Lohmann, D., Sincero, J., Schröder-Preikschat, W., 2011. Feature consistency in compile-time-configurable system software: Facing the linux 10,000 feature problem. In: 6th Conference on Computer Systems. ACM, New York, NY, USA, pp. 47–60. http://dx.doi.org/10.1145/1966445.1966451.

The kernel development community, 2022a. Kconfig language. https://docs.kernel.org/kbuild/kconfig-language.html, Accessed 2022/10/17.

The kernel development community, 2022b. Kernel build system. https://docs.kernel.org/kbuild/index.html, Accessed 2022/10/17.

The kernel development community, 2022c. Linux kernel. https://github.com/torvalds/linux, Accessed 2022/10/17; cloned 2022/09/29, 9:00am using https://github.com/torvalds/linux.git.

The kernel development community, 2022d. Linux kernel makefiles. https://docs.kernel.org/kbuild/makefiles.html, Accessed 2022/10/17.

Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G., 2014. A classification and survey of analysis strategies for software product lines. Comput. Surv. 47 (6), 1–45. http://dx.doi.org/10.1145/2580950.

Zhang, B., Becker, M., Patzke, T., Sierszecki, K., Savolainen, J., 2013. Variability evolution and erosion in industrial product lines: A case study. In: 17th International Software Product Line Conference. ACM, New York, NY, USA, pp. 168–177. http://dx.doi.org/10.1145/2491627.2491645.