How Configurable is the Linux Kernel? Analyzing Two Decades of Feature-Model History

ELIAS KUITER, University of Magdeburg, Germany
CHICO SUNDERMANN, TU Braunschweig, Germany
THOMAS THÜM, TU Braunschweig, Germany
TOBIAS HESS, University of Ulm, Germany
SEBASTIAN KRIETER, TU Braunschweig, Germany
GUNTER SAAKE, University of Magdeburg, Germany

Today, the operating system Linux is widely used in diverse environments, as its kernel can be configured flexibly. In many configurable systems, managing such variability can be facilitated in all development phases with product-line analyses. These analyses often require knowledge about the system's features and their dependencies, which are documented in a feature model. Despite their potential, product-line analyses are rarely applied to the Linux kernel in practice, as its feature model still challenges scalability and accuracy of analyses. Unfortunately, these challenges also severely limit our knowledge about two fundamental metrics of the kernel's configurability, namely its number of features and configurations. We identify four key limitations in the literature related to the scalability, accuracy, and influence factors of these metrics, and, by extension, other product-line analyses: (1) Analysis results for the Linux kernel are not comparable, because relevant information is not reported; (2) there is no consensus on how to define features in Linux, which leads to flawed analysis results; (3) only few versions of the Linux kernel have ever been analyzed, none of which are recent; and (4) the kernel is perceived as complex, although we lack empirical evidence that supports this claim. In this paper, we address these limitations with a comprehensive, empirical study of the Linux kernel's configurability, which spans its feature model's entire history from 2002 to 2024. We address the above limitations as follows: (1) We characterize parameters that are relevant when reporting analysis results; (2) we propose and evaluate a novel definition of features in Linux as a standardization effort; (3) we contribute TORTE, a tool that analyzes arbitrary versions of the Linux kernel's feature model; and (4) we investigate the current and possible future configurability of the kernel on more than 3,000 feature-model versions. Based on our results, we highlight eleven major insights into the Linux kernel's configurability and make seven actionable recommendations for researchers and practitioners.

CCS Concepts: • **Software and its engineering → Software product lines**; *Software evolution*; • **Theory of computation →** *Automated reasoning*.

Additional Key Words and Phrases: Linux kernel, feature modeling, software product lines, software variability

ACM Reference Format:
Elias Kuiter, Chico Sundermann, Thomas Thüm, Tobias Heß, Sebastian Krieter, and Gunter Saake. 2025. How Configurable is the Linux Kernel? Analyzing Two Decades of Feature-Model History. 1, 1 (April 2025), 45 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' Contact Information: Elias Kuiter, kuiter@ovgu.de, University of Magdeburg, Germany; Chico Sundermann, chico.sundermann@tu-braunschweig.de, TU Braunschweig, Germany; Thomas Thüm, thomas.thuem@tu-braunschweig.de, TU Braunschweig, Germany; Tobias Heß, tobias.hess@uni-ulm.de, University of Ulm, Germany; Sebastian Krieter, sebastian.krieter@tu-braunschweig.de, TU Braunschweig, Germany; Gunter Saake, saake@ovgu.de, University of Magdeburg, Germany.

1 Introduction

"[In 1999,] Linux has millions of users, thousands of developers, and a growing market. It is used in embedded systems; it is used to control robotic devices; it has flown on the space shuttle. I'd like to say that I knew this would happen, that it's all part of the plan for world domination. But honestly this has all taken me a bit by surprise."

— Linus Torvalds (1999), creator and lead developer of the Linux kernel [172]

Today, the open-source operating system Linux plays a huge role throughout all of computing, as it is widely applied in diverse environments [38] (e.g., PCs, smartphones, servers, or cars). To adapt to all these different requirements, the kernel of Linux is organized as a feature-oriented software product line [13, 14, 149]; that is, each distinctive, user-visible characteristic in the kernel constitutes a *feature* [87], which can be selected or deselected in a *configuration*. By only compiling the code associated with selected features, unneeded functionality is omitted in the deployed configuration, which may improve its performance, security, or energy efficiency [70, 80, 148]. On the one hand, this variability makes the Linux kernel very adaptable, which is likely to have contributed to its success story. On the other hand, variability also introduces many new issues in the kernel (e.g., inconsistencies [161], broken configurations [124], configuration mismatches [55], and feature interactions [1]), which are difficult and costly to find, debug, and fix manually.

Kernel practitioners (such as developers, maintainers, or end users) are, to some degree, aware of several such variability-related issues in Linux. For instance, developers discuss the difficulty of configuring Linux distributions for end users,[1] the negative impact of introducing too many features on code readability,[2] how to maintain default configurations[3] as well as decide on default values,[4] and the relevance of feature location [52].[5] While many issues can be resolved with proper discussion, the kernel's configuration language KConfig [37], its tools, and the complexity of the kernel itself are also repeatedly criticized, with developers commenting: "There are simply already far too many [features] and they make the kernel harder and harder to change,"[6] "the config subsystem has grown so large that it's gotten out of control,"[7] and "the config system is a nightmare."[7] *Linux Weekly News* summarizes the situation as follows:

"The kernel's configuration system can be challenging to deal with; Linus Torvalds recently called it "one of the worst parts of the whole project". [...] But it is also a part that nobody is really working on; it receives a bit of maintenance, but there does not appear to be any significant effort out there to address its shortcomings. Two-hundred companies support work on each kernel development cycle, but none of them see the configuration system as one of the problems that they need to solve. Until that changes, we are likely to continue to see users struggling with it."

— Jonathan Corbet (2017), co-founder and executive editor of Linux Weekly News[7]

To facilitate dealing with challenging configuration systems like the Linux kernel's and support users struggling with it, researchers have proposed a multitude of *automated analyses* for software product lines over the past two decades, which can facilitate the management of variability in all development phases [18, 101, 156, 167]. In particular, product-line analyses have been used for product configuration [41, 72, 138], feature-model evolution [122, 168] and slicing [4, 141],

[1]https://lwn.net/Articles/507239/
[2]https://lkml.org/lkml/2008/5/1/65
[3]https://lwn.net/Articles/391372/
[4]https://lkml.org/lkml/2020/7/10/1261
[5]https://lkml.org/lkml/2012/1/6/354
[6]https://lkml.org/lkml/2008/4/30/327
[7]https://lwn.net/Articles/733405/

computation of implicit constraints [11], parsing [89], dead-code analysis [161], code simplification [177], type and model checking [35, 43, 105, 164], dataflow analyses [105], testing and sampling [32, 109, 121, 176], optimization of non-functional properties [148], refactoring [58], and economical estimations [36]. Many of these product-line analyses require that all features and potential dependencies between them are known, for example by documenting them in a *feature model* [87, 167]. As dependencies between features are often complex and difficult to fully express otherwise [91], feature models are typically represented as *propositional formulas* [17, 18, 112, 153]. These feature-model formulas are then analyzed with off-the-shelf solvers, such as *satisfiability (SAT) solvers* [17, 112] or model counters (i.e., *#SAT solvers*) [97, 153]. Despite SAT and #SAT being NP- and #P-complete problems [85, 174], respectively, solvers can efficiently analyze many feature models of varying complexity [81, 103, 112, 153].

In principle, the Linux kernel is a prime candidate for developing and applying product-line analyses, as it is a highly relevant, long-lived software system that has a lot of variability to analyze and understand [86, 107, 135]. However, despite their potential benefits, product-line analyses are rarely applied to Linux in practice [37, 61, 149]. While this is also a matter of adoption and knowledge transfer, computing any product-line analysis for Linux is still technically difficult due to two major challenges: *scalability* and *accuracy*. First, the feature model of Linux is often assumed to be one of the most complex and challenging feature models known to researchers [103, 152, 153]. This complexity severely limits the scalability of many product-line analyses based on consistency (SAT) [89, 94, 168], cardinality (#SAT) [130, 153], enumeration (AllSAT) [18, 63], algebraic operations [3, 4, 6], and knowledge compilation [76, 155, 157, 166]. In addition, researchers often favor analyzing smaller product lines over the Linux kernel [152], although this might provide fruitful insights to both kernel developers and researchers. Second, the feature model of Linux is documented in the configuration language KConfig [37], which has complex semantics that are fully utilized and even occasionally extended for Linux [59, 61, 124, 133, 145]. As there is no straightforward, unambiguous translation of KConfig to propositional formulas, a variety of techniques for extracting feature-model formulas from KConfig specifications have been proposed [21, 59, 61, 88, 123, 150, 161, 178, 181]. Each of these techniques makes different trade-offs regarding the implemented semantics, which affects the accuracy of product-line analyses [54, 61, 123]. Even the `kconfig-sat` project,[8] which envisions to integrate a SAT solver into KConfig, does not fundamentally address the challenges of scalability and accuracy.

These challenges with analyzing the feature model of Linux are best exemplified by examining what we currently know about its configurability. The *configurability* of a product line is a fundamental measure for its amount of variability, which assesses the complexity of a product line and serves as ground truth for making management decisions [18, 78, 98, 161] (e.g., to decide whether a patch introduces too much variability and should be split or rejected).[2] Configurability can be quantified using different metrics, most commonly the number of features [16, 24, 78, 153] and the number of configurations [18, 42, 153, 156, 175] of the product line's feature model. First, the *number of features* intuitively corresponds to the number of variables in the corresponding feature-model formula. This metric is so influential and fundamental that it is stated in dozens of publications, for example to judge the complexity and relevance of product lines, compare them with each other, and learn from their evolution history; we give various Linux-related examples in Table 1. Second, the *number of configurations* intuitively corresponds to the feature-model formula's number of satisfying assignments. This metric has similar applications as the number of features and can additionally be used for fine-grained scoping and reducing variability [7, 139] as well as marketing [156] (e.g., to outshine competitors). It is

[8]https://kernelnewbies.org/KernelProjects/kconfig-sat

Table 1. Configurability metrics for the mainline Linux kernel as reported in the literature and in this paper.

Year	Revision	Architecture	Reported #Features		Reported #Configurations	
			In the Literature†	In this Paper★	In the Literature‡	In this Paper★
2002	`v2.5.45` (earliest)	∗	—	2,916	—	10^{424}
2005	`v2.6.12`	`i386`	3,284 [107]	2,546	—	10^{566}
	`v2.6.28`	`x86`	5,321 [147] 5,323 [22] 6,888 [138]	4,250	—	—
	`v2.6.28`	`x86?`	5,426 [146] 5,701 [12] 6,888 [74, 84, 103]	4,250	—	—
	`v2.6.28?`	`x86?`	6,888 [180]	4,250	—	—
2009	`v2.6.32`	`x86`	6,319 [107] 6,320 [21] 60,072 [103]	4,843	—	—
2010	`v2.6.33`	`x86`	6,467 [78, 91] 6,559 [118] 6,918 [105] 62,482 [103]	4,980	—	—
	`v2.6.33?`	`x86?`	5,913 [15]	4,980	—	—
2011	`v3.1`	∗	11,691 [48]	11,383	—	$> 10^{616}$
2015	`v4.0`	`x86`	11,135 [178]	8,104	—	—
2016	`v4.4`	`x86`	15,500 [55]	8,682	—	—
	`v4.4`	∗	14,607 [51]	15,233	—	—
2017	`v4.13.3`	`x86`	12,797 [8]	10,136	—	—
2018	`v4.18`	`x86`	13,379 [99] 22,352 [99]	10,747	—	—
2020	`v5.8`	`x86`	14,817 [8]	12,335	—	—
2024	`v6.11` (latest)	∗	—	20,024	—	—

— Not reported ? Revision or architecture not clearly indicated ∗ Union (for features) and sum (for configurations) over all architectures

† The number of reported features in the literature was originally collected by Kamali et al. [86]. We extend it with several new data points.

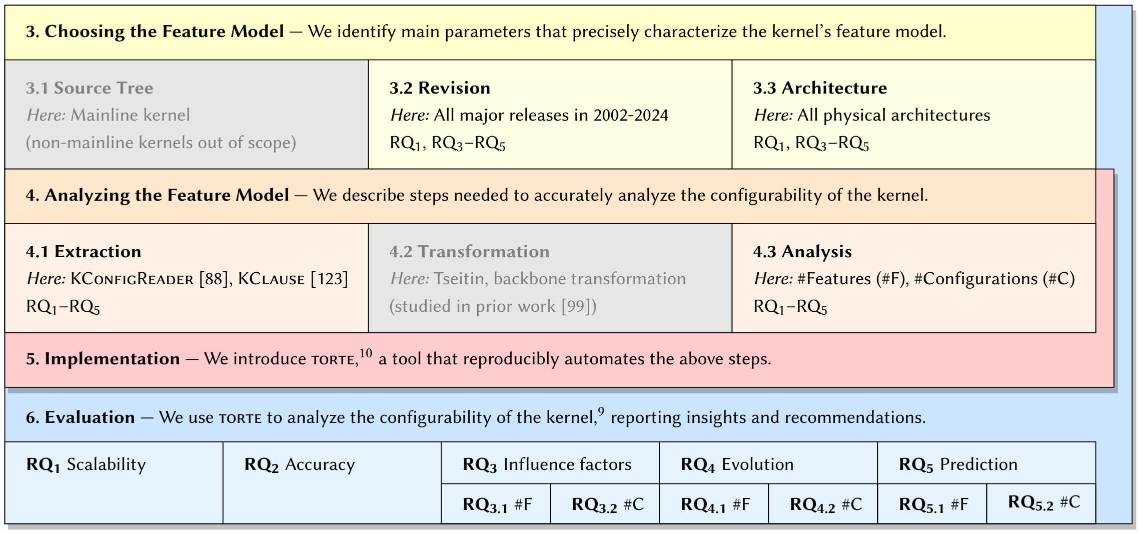
‡ We are not aware of any other publications that report the number of configurations of the Linux kernel.

★ We report results from Section 4 as obtained with KClause (cf. Figure 4, KConfigReader omitted for brevity).

increasingly being reported because the #SAT solvers needed for its computation have made significant progress in recent years [99, 130, 153].

It is crucial to have comprehensive and accurate knowledge about these configurability metrics for two major reasons: First, the number of features and configurations are to variability what source lines of code (SLOC) are to the size of a codebase—fundamental metrics that are widely used to communicate the size of product lines. Having incomplete or inaccurate results for these metrics may lead to flawed management decisions by practitioners, wrong reports in science communication and education [100], and it poses a threat to validity in research evaluations. Second, both of these metrics are closely connected to most advanced product-line analyses [18, 101, 156, 167], which either rely on the product line's set of features (which is reflected in the number of features) or their dependencies (which is reflected in the number of configurations). Thus, any inaccuracies in configurability metrics also point to potential inaccuracies in these analyses, which may lead to flawed management decisions and threaten validity of research evaluations on an even larger scale.

Unfortunately and despite their importance, our current knowledge about these fundamental configurability metrics for the Linux kernel is severely limited. To illustrate this, we show how more than twenty publications report configurability metrics for the Linux kernel in Table 1. We want to highlight four key limitations from the current state-of-the-art in the literature: First, we only found a small number of feature models of the Linux kernel for which feature-model formulas have actually been extracted and analyzed. In particular, no feature models (and, thus, configurability metrics) are available for recent releases of Linux, and no consistent feature-model history is available for applying evolutionary analyses [92, 122, 168, 170]. Second, some publications report a number of features without also reporting the revision and architecture of the feature model. However, both parameters are necessary to precisely characterize the feature model. Third, for a given revision and architecture, the reported numbers of features often disagree. This suggests that there is no consensus among researchers yet about what does or does not constitute a feature in the Linux kernel, although this is a fundamental research question [20]. Moreover, the differences in the metric imply that the underlying



3. Choosing the Feature Model — We identify main parameters that precisely characterize the kernel's feature model.

3.1 Source Tree
Here: Mainline kernel
(non-mainline kernels out of scope)

3.2 Revision
Here: All major releases in 2002-2024
RQ_1, RQ_3–RQ_5

3.3 Architecture
Here: All physical architectures
RQ_1, RQ_3–RQ_5

4. Analyzing the Feature Model — We describe steps needed to accurately analyze the configurability of the kernel.

4.1 Extraction
Here: KConfigReader [88], KClause [123]
RQ_1–RQ_5

4.2 Transformation
Here: Tseitin, backbone transformation
(studied in prior work [99])

4.3 Analysis
Here: #Features (#F), #Configurations (#C)
RQ_1–RQ_5

5. Implementation — We introduce torte,[10] a tool that reproducibly automates the above steps.

6. Evaluation — We use torte to analyze the configurability of the kernel,[9] reporting insights and recommendations.

RQ_1 Scalability	**RQ_2** Accuracy	**RQ_3** Influence factors		**RQ_4** Evolution		**RQ_5** Prediction	
		$RQ_{3.1}$ #F	**$RQ_{3.2}$** #C	**$RQ_{4.1}$** #F	**$RQ_{4.2}$** #C	**$RQ_{5.1}$** #F	**$RQ_{5.2}$** #C

Fig. 1. Structure of this paper's main Sections 3–6, showing how our contributions and research questions relate to each other.

sets of features are also different, which leads to flawed results of various product-line analyses as outlined above. Fourth, only the number of features is reported—we are not aware of any publication reporting the number of valid configurations for Linux, even though this metric is commonly used for other product lines [99, 153]. While computing this metric is a #P-complete problem that is known to challenge current #SAT solvers [153], it is too useful to ignore, as it can be communicated and marketed easily, respects dependencies between features, and is needed for advanced analyses (e.g., feature prioritization [156] or uniform random sampling [130]).

In this paper, we address these four limitations, having two major goals in mind: First, we seek to improve our understanding of variability in the Linux kernel by empirically studying the long-term evolution of two fundamental configurability metrics (i.e., the number of features and configurations) by means of SAT and #SAT solving. Second, we aim to investigate the scalability, accuracy, and influence factors of advanced product-line analyses, which often rely on the features and configurations measured by these configurability metrics. Thus, we aim to establish comprehensive and accurate ground-truth knowledge, so researchers and practitioners can give accurate accounts of the kernel's configurability and make informed decisions based on it.

In particular, we make the following contributions in this paper, grouping them by the four limitations we identified above. In Figure 1, we additionally outline how each contribution is reflected in the structure and contents of this paper.

- **Relevant parameters for analyzing the kernel's feature model are not fully specified, which makes analysis results hard to compare.** To mitigate this, we fully characterize the feature model of the Linux kernel with the parameters *source tree*, *revision*, and *architecture* (cf. Section 3). Thus, we aim to help in choosing the right feature model for a given analysis use case. We also raise awareness about the relevance of these parameters when reporting analysis results. Finally, we evaluate and discuss the influence of the revision and architecture on two configurability metrics (cf. Section 6).
- **The results of product-line analyses disagree, and there is no consensus on how to define features.** To investigate the remaining influence factors on analysis results, we comprehensively describe how to *extract*,

transform, and *analyze* feature-model formulas for Linux (cf. Section 4). We also propose a novel definition of features and evaluate its accuracy by independently comparing two extractors (cf. Section 6). With this definition, we contribute towards standardizing the set and number of features for the Linux kernel. We hope this makes the number of features as unambiguous a metric as, for example, the source lines of code.

- **Analyzed feature models are outdated, and no consistent history is available.** To make the feature model of Linux more amenable to analysis, we contribute TORTE,[10] a Docker-based tool that reproducibly automates extraction, transformation, and analysis of feature models (cf. Section 5). By means of our tool, we are able to publish the first consistent, two-decade-long history of the feature model of Linux, including recent releases.[9] To the best of our knowledge, this is the first feature-model history of this large timescale, which enables evolutionary analyses on the kernel for the first time. In addition, our tool supports arbitrary source trees, revisions, and architectures of Linux, which allows for more tailored analysis needs.
- **The kernel is perceived as too complex to be analyzed, although we lack reliable knowledge about its configurability.** To improve our understanding of the kernel's configurability, we perform a large-scale evaluation of two configurability metrics (i.e., the number of features and configurations) on more than 3,000 feature models of Linux (cf. Section 6). We fully publish our evaluation in a reproduction package[9] along with our tool.[10] In our evaluation, we pose and answer five research questions to investigate the current and possible future configurability of Linux. Moreover, we seek to shed light on the scalability, accuracy, and influence factors of both metrics and, by extension, other product-line analyses. Based on our investigation, we highlight eleven major insights and make seven actionable recommendations for practitioners and researchers. Overall, we provide a thorough empirical analysis to answer the title question: *How configurable is the Linux kernel?*

Compared to previous studies on the Linux kernel's feature model [50, 51, 86, 96, 107, 125, 127, 135, 146], our work is novel in several substantial ways. First, the majority of these studies address different problems than the four limitations we are concerned with. That is, they focus on analyzing changes [50, 96, 125] or coevolution of artifacts [51, 127] instead of configurability. The remaining studies either motivated our work [86, 135] or pioneered feature-model analysis on Linux [107, 146], but we go significantly further: While Kamali et al. [86] and Rothberg et al. [135] both observed inconsistencies in the number of features, they do not analyze them further [86] or focus on a single influence factor [135] (i.e., the relevance of feature-model formulas). Furthermore, while She et al. [146] extracted the first feature model of Linux and Lotufo et al. [107] analyzed the first real-world feature-model history, they do not consider evolution at all [146] or only a short time frame of a single architecture [107]. In particular, none of the referenced studies investigate the number of configurations or compare it to the number of features, as we do.

Besides the differences in goals, the scope of our work substantially exceeds that of previous studies. Notably, we are the first to investigate the entire history of the Linux kernel's feature model in the time frame 2002–2024, which is more than a decade longer than the typically studied time frame.[11] In particular, we also consider releases that are historic (i.e., pre-Git) and recent (i.e., September 2024), and we make predictions about the kernel's possible future. Second, most studies (excluding ours and Rothberg et al. [135]) either do not rely on feature-model formulas at all [50, 51, 96, 125, 127] or do not use them for identifying features [107, 146], although this is advisable [135]. Third, our study is one of the few that investigates differences between architectures [50, 135], while all other referenced studies do not. Fourth, we empirically study the differences between two extractors of feature-model formulas, which has not been done before.

[9]https://zenodo.org/doi/10.5281/zenodo.8190055
[10]https://github.com/ekuiter/torte
[11]The time frames studied in the literature are: '08 [146], '05–'09 [107], '08–'12 [127], '05–'13 [125], '11–'13 [135], '11–'14 [50], '13–'16 [51], '05–'22 [96]

2 Background

We begin by describing the Linux kernel, its variability, and how to analyze this variability using feature-model formulas.

2.1 The Linux Kernel

Linux is a general-purpose, Unix-like [137] operating system for computers. It is developed by a large community of ≈ 2,000 active developers (with more than 20,000 contributors overall)[12] as free and open-source software [40] under the General Public License (GPL) 2.0 [38, 68]. The first version of Linux, `v0.01`, has been released in September 1991 as a hobbyist project by its main developer Linus Torvalds [28, 160]. Today, Linux is used on every other smartphone,[13] on more than half of the 500 most powerful supercomputers,[14] and by 40% of professional developers.[15]

Linux consists of two parts: First, the *Linux kernel*[16] constitutes the core of the Linux operating system. It serves as an abstraction layer between system hardware and user applications and thus manages all memory, peripherals, and input/output requests [160]. On a running Linux system, the kernel consists of a single statically-linked executable, typically located at `/boot/vmlinuz`, as well as various *loadable kernel modules*, typically located at `/lib/modules`. These modules can be loaded at runtime to extend the kernel with new functionality like device drivers or file systems. Thus, they mitigate some disadvantages of a monolithic kernel [27, 159]. Second, *Linux distributions* aim to provide fully-functional systems for certain use cases by bundling the Linux kernel with appropriate configuration files and application software [23]. For example, the well-known Ubuntu Desktop distribution [165] integrates the Linux kernel with a package manager (dpkg/apt), a desktop environment (GNOME), and other applications (e.g., LibreOffice and Thunderbird). Another notable example is the Android mobile operating system [136], which ships with a heavily modified Linux kernel.

We focus specifically on the configurability of the Linux kernel, which lies at the core of any Linux distribution.

2.2 Variability in the Linux Kernel

Today's Linux kernel (`v6.11` as of September 2024) is a huge software system with almost 26 million source lines of C code.[17] Given its many application domains, it is typically not desirable to deploy the full code base every time the kernel is needed in a computer system. This is due to several reasons: First, many embedded systems have limited hardware resources (e.g., disk space, RAM, or bandwidth for network booting) and are thus unable to run the complete kernel [2]. Second, many use cases have restrictions on non-functional properties such as performance, security, or energy efficiency [70, 80, 148]. Third, there is not even a "complete" kernel, as some functionality in the kernel is incompatible by design (e.g., 32-bit conflicts with 64-bit, little endian numbers conflict with big endian numbers). Thus, each use case typically requires a slightly different kernel, which has to be carefully configured beforehand. Consequently, there is a large amount of *variability* in the Linux kernel, which is specified in a feature model.

Feature Modeling To manage variability, the Linux kernel is organized as a feature-oriented software product line [13, 14, 149]. That is, each distinctive, user-visible characteristic of the Linux kernel is regarded as a *feature*

[12]https://lwn.net/Articles/936113/
[13]https://www.statista.com/forecasts/997100/ (9,241 participants, December 2024)
[14]https://www.statista.com/statistics/565080/ (June 2023)
[15]https://survey.stackoverflow.co/2022/ (71,771 participants, May 2022)
[16]https://www.kernel.org/
[17]Counted with `cloc` (https://github.com/AlDanial/cloc): `git clone https://github.com/torvalds/linux && cloc linux`

```
config X86_32
  def_bool y
  depends on !64BIT
  # Options that are inherently 32-bit kernel only:
  select ARCH_WANT_IPC_PARSE_VERSION
  # ...

config SMP
  bool "Symmetric multi-processing support"
  help
    This enables support for systems with more than one CPU. If you have a system with only one CPU, say N.
    If you have a system with more than one CPU, say Y.
    # ...

if X86_32
config X86_BIGSMP
  bool "Support for big SMP systems with more than 8 CPUs"
  depends on SMP
  help
    This option is needed for the systems that have more than 8 CPUs.
endif # X86_32
```

Listing 1. Excerpt of the Linux kernel's feature model, which is specified in the configuration language KConfig.

(also known as a configuration option or symbol) [87]. Since 2002, all features in the Linux kernel are specified and documented as a *feature model* [17, 87] using the internally developed configuration language KConfig [37, 54, 145].

In Listing 1, we show an excerpt of the feature model of Linux. In the KConfig file `arch/x86/Kconfig`, several features are declared using the `config` keyword: For example, the kernel can be compiled either for 32-bit (`X86_32`) or 64-bit systems (`64BIT`), and it might or might not support more than one central processing unit (CPU) or even more than 8 CPUs (`SMP` and `X86_BIGSMP`). Because not all features are compatible with each other, the KConfig file also specifies their dependencies: For example, 32- and 64-bit exclude each other, so `X86_32` requires `64BIT` to be deselected (`depends on !64BIT`). Also, support for more than 8 CPUs only makes sense if the kernel already supports more than one CPU, so `X86_BIGSMP` requires `SMP` to be selected (`depends on SMP`).

In addition, KConfig offers constructs for expressing more information about features and their dependencies: For instance, KConfig distinguishes feature types (here, `bool` for features that are either selected or deselected). It also allows developers to specify default values (e.g., `def_bool y` to select a feature by default), `help` texts, as well as `if` and `select` conditions to further express dependencies [37]. All these features, their dependencies, and additional information then comprise the kernel's feature model.

Configuration, Implementation, and Compilation Given this feature model, a developer can configure the kernel by selecting and deselecting features to produce a *configuration* [2]. If the configuration is *valid* (i.e., it fulfills all feature dependencies), the build system chooses all files that need to be compiled, which are specified in the `Makefile` dialect KBuild [34, 65]. Then, the C preprocessor [66, 67] chooses all lines of code in the selected files that are related to features selected in the configuration [104]. Finally, all preprocessor-selected lines in all KBuild-selected files are compiled into a single executable, accompanied by several modules [96, 115].

In this paper, we focus on the feature model of the Linux kernel (as specified with KConfig), which is necessary (if not always sufficient) for analyzing variability in the kernel.

2.3 Automated Analysis of Feature Models

While the KConfig language is essential for specifying features and their dependencies in the Linux kernel, the capabilities of its tooling are severely limited with regard to feature-model analysis [54, 124]. This is because its parser is simplistic and cannot reason about dependencies beyond checking whether a configuration is valid. Thus, many feature-model and product-line analyses [18, 109, 156, 167] operate on *propositional formulas* [13, 17, 117, 140], which are well-suited for computing analyses on typical feature models [44, 81, 103, 112, 153].

A *feature-model formula* is a pair (F, ϕ), where:

- F is a set of *features*, each of which can be either selected or deselected, for example:

$$F = \{\texttt{X86\_32}, \texttt{64BIT}, \texttt{SMP}, \texttt{X86\_BIGSMP}\} \qquad \text{(features for Listing 1)}$$

- ϕ is a propositional (i.e., Boolean) formula that expresses all *dependencies* between the features in F, for example:

$$\phi = (\texttt{X86\_32} \rightarrow \neg\texttt{64BIT}) \wedge (\texttt{X86\_BIGSMP} \rightarrow (\texttt{X86\_32} \wedge \texttt{SMP})) \qquad \text{(formula for Listing 1)}$$

  All propositional variables mentioned in ϕ must be features; that is, $Var(\phi) \subseteq F$. The remaining features $F \setminus Var(\phi)$ are *unconstrained*; that is, they are freely (de-)selectable.

For a given feature-model formula (F, ϕ), a *configuration* of selected features $C \subseteq F$ is *valid*[18] when it fulfills all feature dependencies in ϕ; that is, evaluating the formula ϕ at the configuration C yields `true` (denoted as $\top$):

$$\phi(C) \coloneqq \phi^{[\forall f^+ \in C:\quad f^+ \rightarrow \top]}_{[\forall f^- \in F \setminus C:\, f^- \rightarrow \bot]} \Leftrightarrow \top \qquad \text{(validity of a configuration } C\text{)}$$

We can compute the aforementioned analyses on a feature-model formula with off-the-shelf solvers, such as *satisfiability (SAT) solvers* [53, 116] or model counters (i.e., *#SAT solvers*) [60, 171]. Two fundamental analyses are *consistency* [18] (i.e., "is the model free of errors?") and *cardinality* [156] (i.e., "how many configurations does it have?"):

$$[\![\phi]\!]_F \coloneqq \{C \subseteq F \mid \phi(C) \Leftrightarrow \top\} \qquad \text{(configuration space of } \phi\text{)}$$

$$SAT(\phi) \coloneqq [\![\phi]\!]_{Var(\phi)} \neq \emptyset \qquad \text{(consistency: whether } \phi \text{ is satisfiable)}$$

$$\#SAT(F, \phi) \coloneqq |[\![\phi]\!]_F| \qquad \text{(cardinality: number of valid configurations for } \phi\text{)}$$

While straightforward in principle, this approach has caveats: First, most cutting-edge SAT and #SAT solvers require the formula ϕ to be supplied in *conjunctive normal form (CNF)*; that is, as a conjunction ($\wedge$) of disjunctions ($\vee$) of literals (F or $\neg F$) [25, 83, 99]. Second, each unconstrained feature in $F \setminus Var(\phi)$ doubles the number of configurations because it is freely (de-)selectable, so F must be passed as an argument to *#SAT* [45] to get correct analysis results. Third, most publications assume that there are only *Boolean* features, which are either selected or deselected. However, $\approx 4\%$ of the kernel's features are *non-Boolean* (e.g., numerical) [21, 126], which can skew analysis results [9, 114, 117]. We discuss these issues in more detail in Sections 4 and 6.

We focus on feature-model formulas with Boolean features, which can be analyzed with state-of-the-art solvers.

[18] In this paper, we only consider valid configurations, so we mostly omit the qualifier "valid".

3 Choosing the Feature Model

Our overall goal is to investigate the configurability of the Linux kernel by extracting, transforming, and analyzing its feature-model formulas, which describe all of Linux's features and their dependencies. However, before we can extract such formulas, we first have to specify accurately *which* Linux kernel we are referring to. This step is sometimes overlooked in the literature, which explains seemingly contradictory analysis results (cf. Table 1) [86, 135].

We identified three main parameters that characterize the feature model of the Linux kernel and can therefore affect all analysis results: the *source tree*, *revision*, and *architecture* of the Linux kernel. In the following, we describe each parameter with examples, use cases, and the parameter values we consider in our evaluation.

3.1 Source Tree

There are many different *source trees* (e.g., forks of Git repositories) for the Linux kernel. Each recent source tree includes KConfig files that define the kernel's feature model. So, choosing an appropriate source tree is the first step for extracting a feature-model formula.

The canonical source tree is the *mainline kernel*, which is maintained by Linus Torvalds on `kernel.org` [39].[19] Since 2005, this source tree is managed with the version control system Git, which Linus Torvalds developed specifically for this use case. The mainline source tree is to Linux what the `main` branch usually is to other software projects: It is the canonical source tree for obtaining, building, and deploying the kernel. Thus, the ultimate goal of any kernel developer is for their patches to be accepted into the mainline kernel. Consequently, the mainline kernel is a natural choice for extracting "the" feature-model formula for Linux.

Still, considering non-mainline source trees for extracting feature-model formulas may also be sensible: First, kernel development is organized using a lieutenant system; that is, the management of subsystems is delegated to trusted experts, each of which usually maintains their own so-called *next tree* [39]. Such next trees (e.g., `linux-next`) contain patches aimed to be included in the next mainline release. Thus, they are comparable to `nightly` branches elsewhere and can be used to extract bleeding-edge feature-model formulas. For example, maintainers can assure the quality of their submitted patches by extracting a feature-model formula from their own next tree and checking it for newly introduced anomalies (e.g., unmet dependencies [124] or dead features [21]). Second, recent bug fixes are often backported to older Linux revisions and then released on the *stable tree* [39].[20] Extracting feature-model formulas from the stable tree can be useful when analyzing kernels that are currently in broad use, as these are usually not available in the mainline. Third, while kernel developers are generally encouraged to merge their patches into the mainline, hardware vendors and software distributors often have their own source trees with custom patches. For example, the Ubuntu kernel team maintains their own source trees for each Ubuntu release.[21] These vendor-specific source trees may include ad-hoc bug fixes, proprietary modules with non-GPL licenses, or configuration patches that select certain features. By extracting feature-model formulas from such source trees, more specific analysis use cases become possible (e.g., to determine how much variability is bound by the Ubuntu kernel compared to the mainline kernel).

Like previous publications (cf. Table 1), we focus on the mainline kernel in this paper, which is the canonical and up-to-date source tree of the Linux kernel.

[19] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/ (Mirror: https://github.com/torvalds/linux)
[20] https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/
[21] e.g., https://git.launchpad.net/~ubuntu-kernel/ubuntu/+source/linux/+git/jammy (Documentation: https://wiki.ubuntu.com/Kernel/FAQ/UbuntuDelta)

3.2 Revision

Each source tree of the Linux kernel typically has a large number of *revisions* (also known as versions or Git commits) [10]. For instance, the mainline kernel has over 1,300,000 revisions, with over 800 tagged revisions (e.g., `v6.11`, `v6.12-rc1`) and over 100 major releases (i.e., tags excluding `-rc` release candidates and other minor releases).[22] Potentially, each revision may have a different feature model. In practice, however, only ≈ 4% of all commits touch any KConfig file and, thus, the feature model [96]. Also, kernel developers are encouraged to separate their patches into small, logically independent changes, which are merged separately [39]. Thus, most patches can be expected to only make small changes to the feature model.

For extracting a feature-model formula, the most recent revision of the kernel is typically the most relevant. To avoid referring to a temporary or unstable state of the kernel, one can use the latest tagged, non-`rc` revision (e.g., `v6.11`). [50]

However, there are also use cases involving feature models for other revisions: First, choosing an untagged revision may be sensible to analyze the effects of bleeding-edge patches (e.g., on a non-mainline kernel). Second, choosing an outdated revision can help with approximating the results of complex feature-model analyses that do not scale to the latest kernel revision. Third, by choosing several revisions (sampled in regular time intervals), the historic development of the Linux kernel can be analyzed.

With regard to the last use case, the oldest commit recorded in the mainline kernel's Git repository is from April 2005 (`v2.6.12-rc2`), when Git was developed specifically for the Linux kernel. A pre-Git history is available on `kernel.org`.[23] When extracting historic feature-model formulas for Linux, this should be kept in mind, as KConfig files have been introduced in 2002 (`v2.5.45`), when Git did not exist yet. Before revision `v2.5.45`, features were specified in a pre-KConfig configuration language that mostly acts as a subset of the `bash` language. While this language can be theoretically analyzed as well, we are not aware of any tools that allow to do so. Thus, the year 2002 is the starting point for a practical analysis of Linux' feature-model history.

We consider all tagged, non-`rc` revisions of the Linux kernel since 2002, as we want to learn from its history.

3.3 Architecture

Given a source tree and revision of the Linux kernel, there is always a number of *architectures* to choose from. By selecting an architecture, the kernel is targeted towards a specific class of CPUs; that is, a kernel built for an `x86` CPU cannot be run on an `arm64` CPU. In Table 2, we show all architectures currently supported by the Linux kernel (i.e., all directories in `arch/`), as well as all subsumed and discontinued architectures. Some architectures may themselves be split into several "sub-architectures" (e.g., 32- and 64-bit for `x86`), which we do not list. We list each architecture's major vendors (e.g., original developer or current manufacturer) as documented in the kernel, online, or in the literature [27].

While no precise market shares for each architecture are known, it is clear that currently, AMD and Intel dominate the market for personal computers with the `x86` architecture,[24] while ARM dominates the market for smartphones and tablets with almost 8 billion licensed unit shipments running the `arm` or `arm64` architecture.[25] Thus, both are natural choices for extracting relevant feature-model formulas.

[22]All statistics in Section 3 have been extracted in September 2024.
[23]https://git.kernel.org/pub/scm/linux/kernel/git/history/history.git/
[24]https://www.statista.com/statistics/735904/ (Q4 2024)
[25]https://www.statista.com/statistics/1131983/ (Q3 2024)

Table 2. Architectures supported by the Linux kernel in September 2024.

Architecture	Subsumed Architecture	Major Manufacturer or Vendor	Remarks
alpha		DEC, Hewlett-Packard	
arc		ARC International, Synopsys	
arm, arm64	arm26	ARM	Predominant on smartphones and tablets
csky		C-SKY Microsystems	
hexagon		Qualcomm	
loongarch		Loongson Technology	
m68k (m68000)	m68knommu	Motorola	
microblaze		Xilinx, AMD	
mips	mips64	MIPS Technologies	
nios2		Altera	
openrisc		—	Open source (GPL)
parisc	parisc64	Hewlett-Packard	
powerpc	ppc, ppc64	Apple, IBM, Motorola	
riscv		—	Open source (BSD)
s390 (z Systems)	s390x	IBM	
sh (SuperH)	sh64	Hitachi, STMicroelectronics	
sparc	sparc32, sparc64	Sun Microsystems, Oracle	
um (User Mode Linux)		—	Virtual machine, excluded from our evaluation
x86	i386, x86_64	Intel, AMD	Predominant on personal computers
xtensa		Tensilica	

Discontinued architectures: avr32, blackfin, cris, c6x, frv, h8300, ia64, metag, mn10300, m32r, nds32, score, tile, unicore32, v850

However, the other architectures should not be overlooked [48, 50]: For example, the sh and s390 architecture are used in automotive applications and mainframe computers, respectively. As a rule of thumb, most architectures can be expected to have some (possibly niche) use case today, as obsolete or troublesome architectures are regularly identified and removed [7].[26] The um (user mode) architecture is special, as it implements a virtual machine [49].

We consider all non-um architectures in Table 2, as they cover all physical devices the Linux kernel has ever been run on.

3.4 Discussion

After choosing appropriate values for the three discussed parameters, we can easily configure and compile a Linux kernel on the command line. Given a source tree *src*, revision *rev*, and architecture *arch*, we refer to the feature model of the specified Linux kernel as $\mathbf{L}\langle {}^{src}_{rev}/arch\rangle$. We can configure such a feature model $\mathbf{L}\langle {}^{src}_{rev}/arch\rangle$ with:

```
git clone <src> linux && cd linux   # download the kernel's source code
git checkout <rev>                  # select revision
make ARCH=<arch> menuconfig         # run interactive configurator
make && make modules                # compile kernel and modules
```

However, the kernel's built-in tools for KConfig can only validate and construct specific configurations; they cannot compute most feature-model analyses, including fundamental metrics that measure the configurability of the Linux kernel (cf. Section 2). To this end, we extract, transform, and analyze feature-model formulas.

4 Analyzing the Feature Model

To compute feature-model analyses (e.g., configurability metrics) for a feature model $\mathbf{L}\langle {}^{src}_{rev}/arch\rangle$ of Linux, we perform three steps: First, we extract a feature-model formula from the kernel's KConfig specifications. Then, we apply two

[26] e.g., https://lwn.net/Articles/751885/, https://lwn.net/Articles/920259/, and https://lwn.net/Articles/950466/

transformation steps to bring the formula into an appropriate form. Finally, we analyze the resulting formula with an off-the-shelf solver to determine its configurability in terms of number of features and configurations.

4.1 Extraction

In the first step, we extract a feature-model formula for a given feature model $\mathbf{L}\langle {}^{src}_{rev}/arch \rangle$. To this end, we need a KConfig *extractor* that constructs a feature-model formula from the given KConfig specifications. Abstractly, such an extractor is given by a pair of functions $(extract_F, extract_\phi)$, which can compute a feature-model formula $(F_{extracted}, \phi)$:

$$F_{extracted} \coloneqq extract_F(\mathbf{L}\langle {}^{src}_{rev}/arch \rangle) \qquad \text{(extracted set of features)}$$

$$\phi \coloneqq extract_\phi(\mathbf{L}\langle {}^{src}_{rev}/arch \rangle) \qquad \text{(extracted propositional formula)}$$

Here, we focus on four influential KConfig extractors that were used in publications listed in Table 1: Undertaker, LVAT, KConfigReader, and KClause. We omit two more recent and lesser-known additions, KFeature [181] and ConfigFix [61], as the former is only applicable to small, artificial feature models and the latter is a re-implementation of KConfigReader, which we expect to yield similar results.

Undertaker and LVAT Undertaker [161] is a tool for identifying dead code in KConfig-based configurable systems. To this end, it runs a kernel binding named *dumpconf*, which extracts structured information about all features and their dependencies in Rigi Standard Format (RSF) [90]. Undertaker then reads the resulting RSF file and applies the KConfig semantics to translate it into a feature-model formula. Similarly, the Linux Variability Analysis Tools (LVAT) [21] first run a binding to extract feature information, which is then translated into a feature-model formula. Both Undertaker and LVAT have limited accuracy [54], which is mostly due to the complex semantics of KConfig [59, 61, 124, 145].

KConfigReader and KClause KConfigReader [88] is part of the TypeChef infrastructure for variability-aware parsing and type checking [89]. It follows the same approach as Undertaker and LVAT, but uses a modified version of *dumpconf* and has a more accurate translation semantics (e.g., it partially translates non-Boolean features) [54]. Similarly, KClause [123] is part of the KMax tool suite for analyzing KConfig and KBuild dependencies [65]. KClause uses the binding *kextractor* and produces slightly more accurate and compact formulas than KConfigReader [123].

We focus on the extractors KConfigReader and KClause, as they reflect the semantics of KConfig most accurately.

4.2 Transformation

In the second step, we perform a CNF transformation on the extracted formula (which is the required input format of most solvers), as well as a backbone transformation (which is not strictly necessary, but useful, as outlined below).

CNF Transformation Most SAT and #SAT solvers require a formula in conjunctive normal form (CNF) as input (cf. Section 2). Thus, we must transform ϕ into a CNF formula ϕ_{CNF}. Intuitively, we can easily do this by applying the distributive law of propositional logic as well as De Morgan's laws to the formula ϕ [30]. However, for complex feature models like the Linux kernel, this transformation is prone to an exponential explosion of terms [99]. Instead, we apply the CNF transformation proposed by Tseitin [173], which is linear-time in the size of ϕ and abbreviates complex sub-formulas with new auxiliary variables. Due to these auxiliary variables, the resulting CNF formula ϕ_{CNF} is only quasi-equivalent to ϕ (which is a strong version of *equi-satisfiability* [99]). Consequently, we must ignore these auxiliary variables when interpreting analysis results [99]. We compute ϕ_{CNF}, V_{all}, and V accordingly:

$$\phi_{CNF} \coloneqq tseitinCNF(\phi) \qquad \text{(CNF formula obtained with Tseitin transformation [99])}$$

$$V_{all} \coloneqq Var(\phi_{CNF}) \qquad \text{(all variables, including auxiliary)}$$

$$V \coloneqq Var(\phi) \qquad \text{(natural variables, excluding auxiliary)}$$

Previous work on analyzing the feature model of Linux [21, 59, 61, 88, 123, 161, 178] either uses the Tseitin transformation or an optimization by Plaisted and Greenbaum [128]. The preferred transformation depends on the use case: The Tseitin transformation preserves the cardinality of ϕ, so it is suitable for counting configurations [99]. While the Plaisted-Greenbaum transformation is not suitable for counting [61, 99, 123], it is drastically more efficient than the Tseitin transformation when enumerating configurations instead of counting them [108].

Backbone Transformation Subsequently to the CNF transformation, we apply an (optional) backbone transformation to the CNF formula ϕ_{CNF}. The backbone [82] of a formula comprises all of its variables that are either not selectable (i.e., *dead*) or not de-selectable (i.e., *core*). The backbone of ϕ_{CNF} is mathematically defined as follows:

$$V_{dead} \coloneqq \{v \in V \mid \neg SAT(\phi_{CNF} \wedge v)\} \qquad \text{(dead variables)}$$

$$V_{core} \coloneqq \{v \in V \mid \neg SAT(\phi_{CNF} \wedge \neg v)\} \qquad \text{(core variables)}$$

$$\phi_{CNF}^{back} \coloneqq \phi_{CNF} \wedge \bigwedge\nolimits_{v \in V_{dead}} \neg v \wedge \bigwedge\nolimits_{v \in V_{core}} v \qquad \text{(CNF formula with explicit backbone)}$$

In practice, more efficient algorithms for computing the backbone are available, which require fewer SAT calls [26, 82]. We discuss these briefly in Section 6, so we can efficiently compute the backbone of Linux. Based on the explicit backbone in ϕ_{CNF}^{back}, we can then easily infer dead and core features [18], which do not contribute to variability in the kernel. While this backbone transformation is optional, it is useful for identifying the “true” variability in a feature model and accelerating some analyses (e.g., feature-model simplification [73, 143, 177]).

We apply the Tseitin transformation, as we aim to count the number of the kernel’s configurations. We also apply a backbone transformation, as we want to assess the influence of dead and core features.

4.3 Analysis

In the final step, we execute some feature-model analysis (cf. Section 2) by running an off-the-shelf solver on the transformed formula for $\mathbf{L}\langle {}^{src}_{rev}/arch\rangle$. While the exact procedure depends on the analysis, often knowledge about all features in the feature model is required [13, 18]. Intuitively, we might suspect that the set of features is given by $F_{extracted}$ (and the number of features by $|F_{extracted}|$); however, this set and number are not necessarily accurate as explained below. In addition to the set of features, knowing the number of configurations also has useful applications [156] besides being another configurability metric. Similar to above, we might suspect that the number of configurations is given by $\#SAT(V, \phi)$; however, we will show how this is not necessarily accurate either, thus affecting measurements of configurability and other product-line analyses.

In the following, we propose several candidates for both the number of features and configurations, which are fundamental configurability metrics of any product line and widely used in the literature [18, 50, 86, 130, 135, 153, 156]. Thus, we aim to assess the effect of various influence factors on these configurability metrics and also on a wide array of product-line analyses, which often rely on the same underlying data.

Number of Features While KConfigReader and KClause both return a set of extracted features $F_{extracted}$, this set also contains unrelated entries (e.g., regarding modules or non-Boolean features, which only encode specific feature values) and also dead features (which do not contribute to variability). Instead, we propose a novel definition of the set of features that additionally identifies features based on the variables in the feature-model formula ϕ; so, based on the set of natural variables V. Building on both $F_{extracted}$ and V, we show how to derive an accurate set of features F step-by-step, where each intermediate step defines a potential candidate for enumerating and counting features.

First, we start with the natural variables V and remove all its variables that do not directly relate to features. This includes variables that refer to modules (for KConfigReader) and visibility conditions (for KClause).[27]

$$FV \coloneqq \{v \in V \mid v \not\simeq \texttt{MODULE} \land v \not\simeq \texttt{VISIBILITY} \land \dots\} \qquad \text{(natural feature variables)}$$

Next, some variables in FV are not actually selectable in the configurator (e.g., if they are specific to another architecture than *arch*). We remove these dead variables, as they cannot be configured, and intentionally so [75]. We keep core variables, as they represent important functionality that must always be selected.

$$FV_{undead} \coloneqq FV \setminus V_{dead} \qquad \text{(undead feature variables)}$$

Next, some features are not included yet in FV, as they might not have any dependencies encoded in the formula ϕ. We obtain these *unconstrained* features from the extracted features $F_{extracted}$ and add them, arriving at F_{all}.

$$F_{unconstrained} \coloneqq F_{extracted} \setminus V \qquad \text{(unconstrained features)}$$

$$F_{all} \coloneqq FV_{undead} \cup F_{unconstrained} \qquad \text{(all features)}$$

However, both extractors encode non-Boolean features to some degree into V. Thus, its refinement F_{all} still contains entries related to non-Boolean variability, which do not directly correspond to features defined in any KConfig file. To address this issue, we identify all features defined in all KConfig files (i.e., with standard tools like `grep` and `awk`), which yields an "upper bound" F_{max} on the features of $\mathbf{L}\langle {}^{src}_{rev}/arch\rangle$. By intersecting F_{all} with F_{max}, we obtain the set of features F; with the number of features then simply being $\#F \coloneqq |F|$.

$$F_{max} \coloneqq \{f \mid \texttt{config\_}f \simeq \mathit{file} \land \mathit{file} \in \mathit{KConfigFiles}(\mathbf{L}\langle {}^{src}_{rev}/*\rangle)\} \qquad \text{(upper bound on features)}$$

$$F \coloneqq F_{all} \cap F_{max} \qquad \text{(features)}$$

In Section 6, we evaluate the differences between these feature candidates, and whether F is indeed accurate.

Number of Configurations Intuitively, the number of configurations $\#C$ has an exponential upper bound limited by the number of features $\#F$ (i.e., $\#C \leq 2^{\#F}$). However, the number of configurations is usually highly constrained by ϕ (i.e., $\#C \ll 2^{\#F}$), which makes $2^{\#F}$ an inaccurate estimate for the number of configurations [153]. Instead, we calculate a lower bound $\#C_{min}$ by calling a #SAT solver on the transformed feature-model formula ϕ^{back}_{CNF}.[28] Still, this method ignores unconstrained features, which increase the number of configurations further. Thus, we propose to consider these features as well to produce a more accurate number of configurations $\#C$.

$$\#C_{min} \coloneqq \#SAT(V_{all}, \phi^{back}_{CNF}) \qquad \text{(lower bound on \#configurations)}$$

$$\#C \coloneqq \#SAT(V_{all} \cup F_{unconstrained}, \phi^{back}_{CNF}) = \#C_{min} \cdot 2^{|F_{unconstrained}|} \qquad \text{(\#configurations)}$$

[27]With $\simeq$, we refer to a regular expression matcher, and "…" denotes additional conditions. Details on both are available in our reproduction package.[9]
[28]We use ϕ^{back}_{CNF} instead of ϕ_{CNF}, as this typically speeds up the computation.

Again, we evaluate whether there is a relevant difference between both of these candidates in Section 6.

We focus on counting features and configurations, which are fundamental metrics for understanding the configurability of the Linux kernel, besides being prerequisites for computing many other product-line analyses.

4.4 Discussion

To improve the accuracy of basic feature-model analyses, we proposed several feature and configuration candidates. Still, our proposals $\#F$ and $\#C$ may be imperfect in several subtle ways, which we discuss here.

Number of Features First, our definition of FV_{undead} may capture accidentally dead features; that is, features that are not selectable (but intended to be) for *arch*. However, this case rarely occurs in practice [21]. Second, it might be tempting to skip the extraction of a feature-model formula entirely and only report F_{max} (i.e., the features identified by searching all KConfig files). However, F_{max} includes false-positives (e.g., features in KConfig files that are never included [161]). Also, this approach cannot distinguish architecture-specific features [135] and, thus, it makes any non-trivial feature-model analyses difficult. Or, as phrased by Rothberg et al. [135]: "[For] any research that is about facts of variability models in Linux, only a semantic approach makes sense." Third, we do not specifically detect mandatory features in the kernel (which are only implicitly encoded in KConfig files with visibility conditions and default values). However, we do compute all core features, which are mandatory features that cannot be deselected at all. Thus, all remaining mandatory features indeed contribute to the configurability of the kernel, which we are concerned with.

Number of Configurations For counting configurations accurately, a feature-model formula is essential. While encoding Boolean features into the formula is mostly straightforward [13, 17, 140], the Linux kernel also contains *non-Boolean features*: namely, `tristate` (with a third truth value for modules), `int/hex` (numerical), and `string` (free-text) features. Many extractors are aware of this non-Boolean variability [54, 59, 61, 88, 123, 124, 181]. However, most only address `tristate` features and do not fully encode `int/hex` and `string` features, as they only constitute $\approx 4\%$ of features [21] and are difficult to encode propositionally [117]. For counting configurations, however, even a few numerical features may have a huge impact [9, 114, 117]. Because this impact may bias analysis results towards numerical features, it is unclear whether and when including them in the number of configurations is even beneficial. For `string` features, the potential impact is even worse, as they can cause the configuration space to become infinite.

Here, we simply use the extracted formula ϕ as is and rely on the extractor's default behavior, as most publications that use these extractors do as well. For KConfigReader, this means Boolean features are encoded with two truth values and `tristate` features with three truth values. In contrast, KClause encodes `tristate` features with two truth values instead, thus merging the values for static and module compilation. Both extractors encode `int/hex` and `string` features partially by encoding transitive equality dependencies over non-Boolean features, thus underapproximating the number of configurations [124]. For example, suppose that f_{int} is an `int` feature and both $f_1 \rightarrow f_{int} = 1$ and $f_2 \rightarrow f_{int} = 2$ are feature dependencies of f_1 and f_2. In this situation, KConfigReader correctly infers that f_1 and f_2 are exclusive (i.e., they cannot be selected together). Following others [48], we argue that this is a sensible approximation of the Boolean fragment of the kernel's feature mode, as most non-Boolean features only makes small changes to the kernel executable. Moreover, a full encoding of non-Boolean features is not even desirable for some analyses [130].

5 Implementation

Many tools have been proposed for extracting [21, 61, 88, 123, 161, 181], transforming [57, 110], and analyzing [5, 19, 110] feature-model formulas. However, there is no tool yet for consistently computing analyses on entire histories of feature models based on the KConfig language. As we aim to analyze the complete history of the Linux kernel's feature model since 2002, we implemented a new open-source tool, torte (*KConfig Extrac**tor** that **T**ackles **E**volution*),[29] which can deal with the large scale of our evaluation and improves upon existing tools. The purpose of torte is to bundle existing tools (e.g., KConfigReader [88], KClause [123], and Z3 [46]) in a declarative workbench, which can describe and execute reproducible evaluations on feature-model formulas.

With torte, we address two major issues: First, so far only few feature models of the Linux kernel have been successfully extracted (cf. Table 1). This is mostly due to technical limitations in extractors [142] (e.g., KConfigReader has not been updated in eight years and, thus, is not compatible with more recent Linux releases). We removed most of these limitations (which we document in detail in our repository) and are now able to extract feature-model formulas for almost arbitrary source trees, revisions, and architectures of Linux. To this end, we generalized the bindings *dumpconf* and *kextractor*, which we re-compile for every analyzed revision. We confirmed that our modifications are correct by manually comparing our results with those of the original bindings. As for the second issue, analyzing a feature-model formula involves many steps. Especially for large-scale evaluations, these can be tedious to execute manually. To mitigate this issue, we provide full Docker automation for all steps described in Section 4. So, for reproducing our evaluation (or parts thereof), only a single self-executing experiment description file is needed. For example, to extract, transform, and analyze the feature model of Linux `v6.11` on a typical Linux system, a single command suffices.[30] For details on the execution, capabilities, and limitations of our experiment, we refer to the repository of torte.[29]

6 Evaluation

After choosing relevant feature models of Linux (cf. Section 3) and describing how we intend to analyze them (cf. Section 4), we describe and discuss the research questions, methodology, and results of our evaluation.

6.1 Research Questions

In our evaluation, we aim to address the following five research questions concerning the configurability of Linux:

RQ$_1$ For which revisions and architectures can feature-model formulas be extracted, transformed, and analyzed?
RQ$_2$ How similar are the feature and configuration candidates of the extracted feature-model formulas?
RQ$_3$ How do revision, architecture, and extractor influence the number of features (**RQ$_{3.1}$**) and configurations (**RQ$_{3.2}$**)?
RQ$_4$ How do the number of features (**RQ$_{4.1}$**) and configurations (**RQ$_{4.2}$**) evolve in a typical kernel release?
RQ$_5$ How many features (**RQ$_{5.1}$**) and configurations (**RQ$_{5.2}$**) does the Linux kernel have, and how many might it have in the future?

Our first goal with these research questions is to investigate the current and possible future configurability of the Linux kernel by means of two metrics, its number of features and configurations. Based on our investigation, we will highlight major insights and make actionable recommendations for practitioners and researchers. As our second goal, we aim to investigate the general scalability (RQ$_1$), accuracy (RQ$_2$), and influence factors (RQ$_3$) of feature-model analyses on the kernel. As the number of features and configurations are both fundamental metrics of software

[29]https://github.com/ekuiter/torte
[30]`curl -s https://ekuiter.github.io/torte/ | sh -s linux-recent-release`

variability, we expect insights regarding these research questions to also have consequences for other feature-model analyses [18, 109, 156, 167] (e.g., if the number of features is inaccurate, the set of features is as well, which many analyses rely on).

6.2 Methodology

To answer our research questions, we consider the following feature models of the Linux kernel (cf. Section 3):

$$\mathbf{L} \coloneqq \{\mathbf{L}\langle{}^{src}_{rev}/arch\rangle \mid src = \texttt{mainline} \land arch \neq \texttt{um} \land isTagged(rev) \land \texttt{v2.5.45} \leq rev \leq \texttt{v6.11} \land rev \not\approx \texttt{-rc}\}$$

As detailed in Section 3, we focus on the mainline kernel, investigate all CPU architectures (except for the virtual machine `um`), and consider all major tagged revisions of the Linux kernel from 2002 to 2024. Using torte, we run both KConfigReader and KClause on each feature model in **L**, apply the Tseitin and backbone transformation to all extracted feature-model formulas, and compute every feature and configuration candidate proposed in Section 4. We evaluate both KConfigReader and KClause, as both are comparably accurate [123] and actively used by researchers to prepare patches for the Linux kernel [1, 55, 124, 161].

With this approach, we first reduce selection bias due to only evaluating one tool. Second, evaluating both tools allows us to assess the difference in semantics that they assign to non-Boolean features (cf. Section 4). For testing significance (RQ_2) and linear correlation (RQ_3, RQ_4), we use the Wilcoxon signed-rank test and Pearson correlation coefficient, respectively. We publish our evaluation in a reproduction package that includes all feature models and a Python script for generating interactive versions of our figures.[9]

Solvers For applying the CNF transformation, we use the `tseitin-cnf` tactic implemented in the SMT solver Z3 [46], which is optimized to perform well on industrial instances [99]. For extracting backbones, we initially tried Janota's algorithm [82] with Kissat [25], the winning solver in the SAT competition 2022 [83].[31] As this approach takes hours on recent revisions of the kernel, we settle on using CadiBack [26] instead, which is an optimized backbone extractor that performs the same computation in minutes. For counting configurations, we consider the four most efficient #SAT solvers identified in the model-counting competition 2022.[32] We omit `ExactMC`, whose submitted version is not available, as well as `SharpSAT-TD`, which crashes on our data. Thus, we use the remaining #SAT solvers `SharpSAT-td+Arjun` and `d4` as submitted to the competition. By using two solvers, we can cross-validate their results to reduce selection bias. We also considered the approximate #SAT solver `ApproxMC` [33], but it did not perform well in a previous study [153], and it also timed out in all our pre-experiments.

Execution We execute our evaluation on an Intel Xeon E5-2630 2.4 GHz server with 1 TiB RAM. We parallelize jobs in each stage of the evaluation (e.g., by extracting 16 feature models in parallel) to reduce the experiment's total runtime. For #SAT solvers, we set a one-hour timeout with no memory limit. Also, to avoid running out of memory, we only run four parallel jobs. As we expect many timeouts to occur in this stage [153], it is impractical to attempt #SAT on each feature-model formula. Instead, we only allow four subsequent #SAT timeouts for a given line of continuous feature-model history $\mathbf{L}\langle{}^{\texttt{mainline}}_{*}/arch\rangle$, as we expect feature models to grow like lines of code [79, 102]. On failure (i.e., after four hours), we skip the remaining formulas of that line of history and treat them as timeouts. This approach is justified as Linux is growing more complex [69, 79] and, thus, also more challenging for #SAT solvers [153].

[31]https://satcompetition.github.io/2022/
[32]https://mccompetition.org/past_iterations

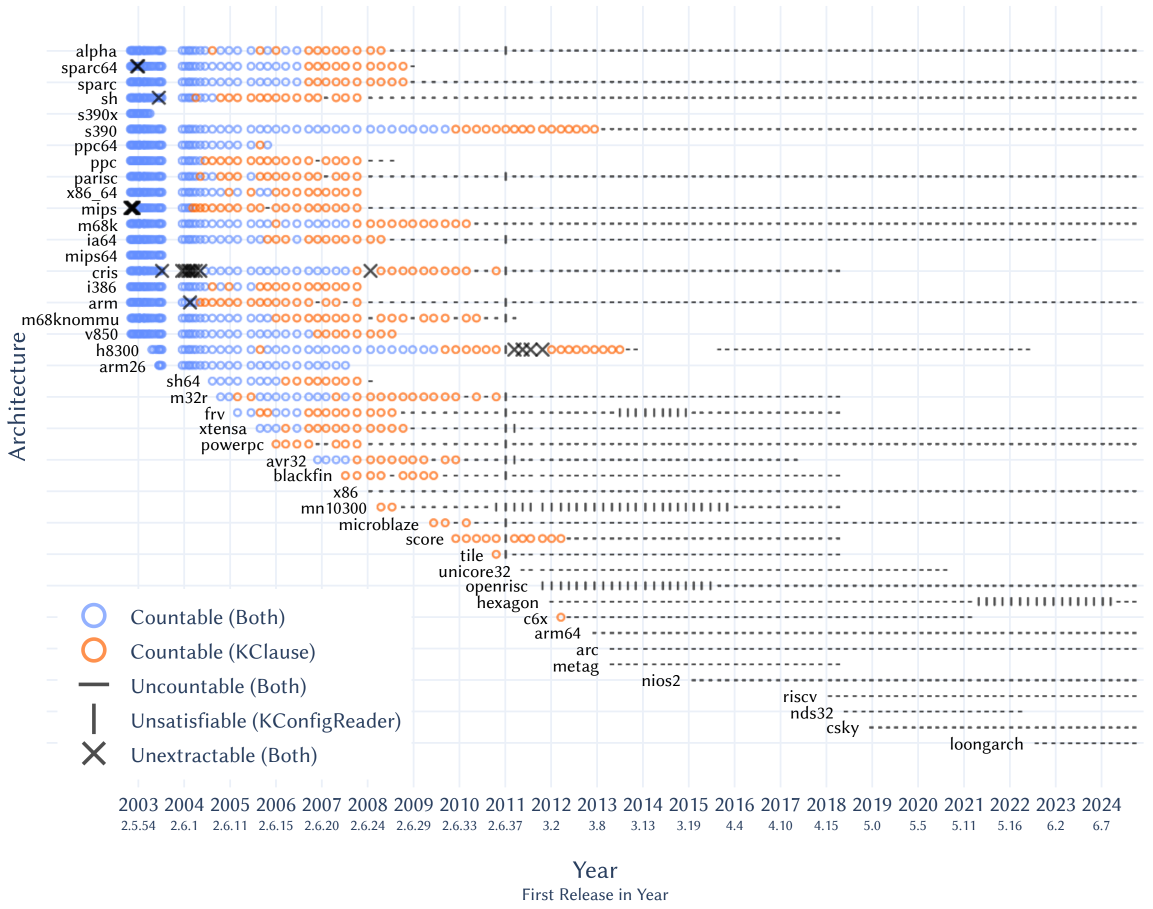


Fig. 2. History of architectures supported by the Linux kernel, including all failures that occurred in our evaluation.

To test the sensitivity of our parameters, we make two independent adjustments, which we only evaluate on the `i386/x86` architecture: First, we increase the #SAT timeout from one hour to six hours; second, we do not skip any feature models for `i386/x86`, thus forcing #SAT calls on the entire history of this architecture.

6.3 Results and Discussion

In the following, we describe and discuss the results of our evaluation for each individual research question.

6.3.1 RQ$_1$: For which revisions and architectures can feature-model formulas be extracted, transformed, and analyzed? In Figure 2, we show all 3,309 feature models in **L**. On the x-axis, we show all 144 analyzed revisions by year, together with the first major kernel release in that year. On the y-axis, we show all 45 analyzed architectures (in the median, 22 per revision); not considering renamings and subsumptions of architectures (cf. Table 2). We use symbols to mark any failures in the extraction, transformation, or analysis of some feature model (`unextractable` implies `unsatisfiable`, which in turn implies `uncountable`; blanks mark (temporarily) discontinued architectures).

Results Most feature models are extracted successfully, while 23 of 3,309 feature models cannot be extracted by either extractor. There are no failures during the CNF transformation; however, during the backbone transformation, we observe that 93 feature models extracted by KConfigReader are unsatisfiable. Next, we attempt 6,479 calls to each #SAT solver (i.e., 6,479 calls = (3,309 models − 23 unextractable) · 2 extractors − 93 unsatisfiable). Of these calls, 2,229 succeed for at least one and 4,250 calls either time out for both solvers or we skip them due to four preceding timeouts. Of the successful calls, 182 are exclusively due to `SharpSAT-td+Arjun` and 16 due to `d4`. In the 2,031 cases where both solvers succeed, they agree on the cardinality.

By increasing the timeout for the `i386/x86` architecture, nine additional calls succeed (included in all figures). Out of those nine calls, four can alternatively be obtained by not skipping any feature models for that architecture.

Discussion Our results show that the extraction and transformation of feature models mostly succeed, while analysis remains challenging. Regarding extraction, the 23 failures can all be attributed to errors in the KConfig specifications (e.g., inclusions of missing KConfig files, typos, and recursive dependencies). Regarding transformation, we suspect that the 93 failures are due to a known inaccuracy in KConfigReader [123]. Regarding analysis, we count the number of features successfully for all extractable and satisfiable feature models in **L**. However, we are not able to count the configurations of many feature models, which is known to be particularly difficult on large and complex feature models [153]. Whether a feature model is countable depends on the revision, architecture, and extractor: While we cannot count any revision after June 2013, older revisions (e.g., from 2007 and earlier) are mostly countable. Also, some architectures are easier to count than others; for example, we cannot count `i386` beyond 2007, while we count `h8300` until 2013 for at least one extractor. Notably, architecture is a much better predictor for countability (cf. Figure 2) than a feature modelrs number of features or its formula's number of literals: That is, we could count all feature models with less than 2,373 features or 58,910 literals; while we could not count any feature models with more than 3,635 features or 814,689 literals. Consequently, this leaves a gap of more than 1,000 features and 750,000 literals, in which we could count some models but not others. Regarding the extractor, feature models extracted by KClause are easier to count.

Finally, increasing the timeout or not skipping feature models does not substantially increase the number of successful #SAT calls, and it does not reveal any new information regarding more recent kernel revisions.

We extract and transform more than 3,000 feature models of the Linux kernel. We count features successfully, while counting configurations only succeeds on old revisions (i.e., until 2007) and less complex architectures (e.g., `h8300`).

6.3.2 RQ_2: How similar are the feature and configuration candidates of the extracted feature-model formulas? On the left side of Figure 3, we compare feature candidates (cf. Section 4). On the x-axis, we show each candidate, distinguishing the extractor. On the y-axis, we compare each feature candidate to the features F in terms of its Jaccard similarity (i.e., F is the reference value and therefore 100% similar to itself). On the right side of Figure 3, we analogously compare both configuration candidates by computing the ratio $\#C_{min}/\#C$. Each value in a box corresponds to one feature model. In addition, we compare the Jaccard similarity of F and the ratio of $\#C$ obtained with KConfigReader with that of KClause.

Results All feature candidates differ significantly from F ($p < 0.001, d = 0.61$). Comparing the features F obtained with KConfigReader and KClause, they have median Jaccard similarity 97.5% (min = 88.2%, max = 99.3%).[33] For the configuration candidates, $\#C_{min}$ differs from $\#C$ by $3.91 \cdot 10^{-3}$ (min = $3.81 \cdot 10^{-6}$, max = 0.5) for KConfigReader

[33]We report median values in the text and minimum and maximum values in parentheses (min = . . . , max = . . .).

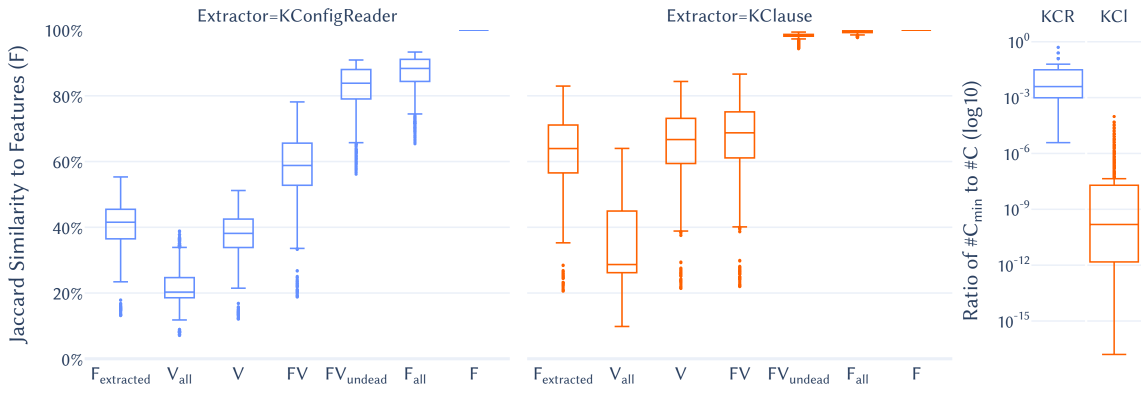


Fig. 3. Comparing feature candidates (on the left) and configuration candidates (on the right) as defined in Section 4.

and $1.53 \cdot 10^{-10}$ (min = $1.6 \cdot 10^{-17}$, max = $9.65 \cdot 10^{-5}$) for KClause. Finally, the logarithmic number of configurations $log_{10}(\#C)$ obtained with KConfigReader is 49.5% larger than that of KClause (min = 34.9%, max = 59.8%).

Discussion Our results show that the choice of feature candidate has a large impact, partially explaining the disagreement in the literature (cf. Table 1). For instance, the extracted features $F_{extracted}$ of KConfigReader typically differ from the features F by 58%. Also, eliminating auxiliary variables from V_{all} is crucial to obtain correct results (e.g., this is why some publications report more than 60,000 features for Linux, mistaking features for auxiliary variables). In addition, we find that omitting all dead features with FV_{undead} has a large impact for both extractors, although this step is rarely considered [135].[34] We argue that the features F are a reasonable reference value, as they are quite similar for both extractors and thus seem most accurate. Regarding the configuration candidates, the unconstrained Boolean and `tristate` features typically have an impact of 3–10 orders of magnitude on the number of configurations, depending on the extractor. For KConfigReader, this impact is smaller because it implements a three-valued semantics for `tristate` features, which creates default constraints for each of them, so they are necessarily constrained. This is consistent with the larger number of configurations obtained with KConfigReader, whose 49.5% longer (i.e., logarithmically larger) number of configurations correspond to the third value of the `tristate` features.[35]

The numbers of features and configurations of the Linux kernel largely depend on the candidates chosen for their measurement. We use $\#F$ and $\#C$, which are the most accurate candidates to date, as per our discussion in Section 4.

6.3.3 *$RQ_{3.1}$: How do revision, architecture, and extractor influence the number of features?*

We show the number of features $\#F$ for each revision and architecture of the Linux kernel in Figure 4b.[36] By computing the union of all architecture-specific features for a given revision,[37] we can determine the total number of features in the Linux kernel,

[34]Regarding core features, we argued in Section 4 for their general inclusion in all feature candidates due to their relevance. However, even if we were to remove them, their influence would be negligible in most cases, as only 0.84% (min = 0.22%, max = 5.97%) of all features are core features.
[35]Switching from two- to three-valued logic, one might expect $\#C$ to be $^{ln\,3}/_{ln\,2} - 1 \approx 58.5\%$ longer than $\#C_{min}$, not 49.5%. However, the former assumes all features to be independent and `tristate`, which they are not, so the actual length difference is smaller.
[36]For each diagram, an interactive version is available in our reproduction package.
[37]We assume here that features sharing names across architectures are identical, which we believe aligns with the intentions of kernel developers.

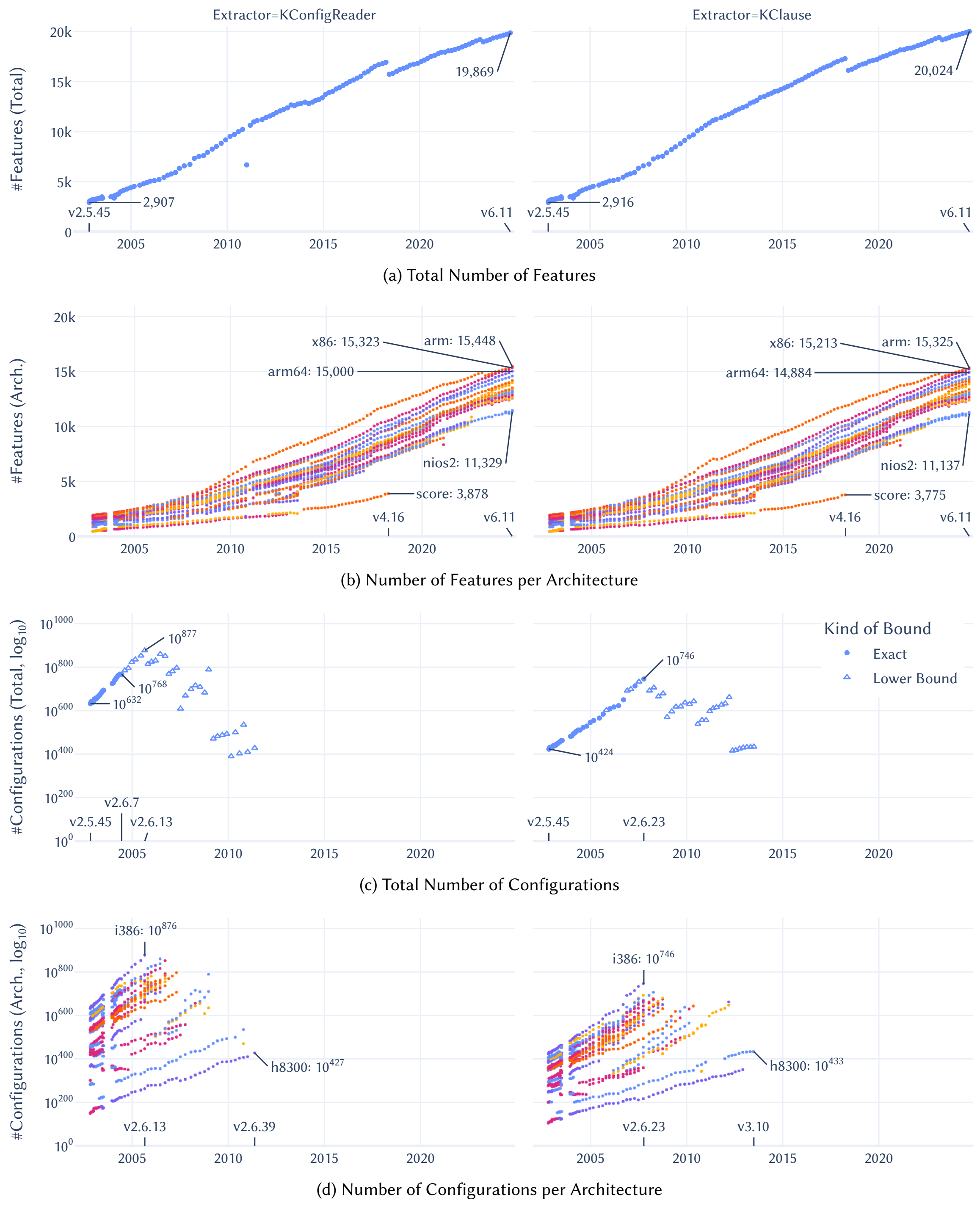


Fig. 4. Number of features (as-is) and configurations (logarithmic) of the Linux kernel, split by extractor.

which we show in Figure 4a. For brevity, we only report numbers obtained with KClause below and in Table 1, as its results are very similar to KConfigReader (RQ_2).

Results The total number of features grows linearly over time for both extractors ($r = 0.99, p < 0.001$). Similarly, we find a slightly lower correlation for each individual architecture with the revision ($r > 0.9, p < 0.001$). The number of features also largely depends on the specific architecture, with 4,936 features in the median (min = 438, max =15,325). Also, the features in a given architecture are 47.6% of the total features (min = 14.7%, max = 76.6%). In Figure 4a, two points in time stand out: First, there is an outlier for KConfigReader in 2011 (`v2.6.37`), which is due to its unsatisfiable feature-model formulas (RQ_1). Second, there is a drop in the total number of features for both extractors in 2018 (`v4.17`), which is due to the removal of several architectures.[26]

Discussion The linear growth of features over time is consistent with the kernel's growth in lines of code [79]. This suggests that Lehman's laws [102] (i.e., software systems continuously grow and increase in complexity), also apply to the complexity and growth of variability in the Linux kernel. Further, the number of features varies for each architecture, which may influence the scalability of product-line analyses [153]. Finally, even the largest architectures (i.e., `x86` and `arm(64)`) only have at most 76.6% of all features. Thus, focusing only on these architectures (as it is typically done, cf. Table 1) overlooks at least one fourth of the features.

The number of features of the Linux kernel grows linearly over time and depends largely on its architecture. When using a suitable feature candidate, it does not depend on the used extractor.

6.3.4 *$RQ_{3.2}$: How do revision, architecture, and extractor influence the number of configurations?* Similarly to $RQ_{3.1}$, we show the total and architecture-specific number of configurations $\#C$ in Figure 4c and Figure 4d, respectively. However, there are some differences to the number of features ($RQ_{3.1}$): First, we report all numbers logarithmically (log_{10}), as the number of configurations is typically exponential in $\#F$ [153]. Second, we compute the total number of configurations by summing the configurations for each architecture (not with a union). Because not all #SAT calls succeed (RQ_1), we distinguish exact results (●) from lower bounds (△) due to #SAT timeouts. Third, we report results for both extractors, as their results differ (RQ_2). We calculate correlations on the logarithmic exact results.

Results The total number of configurations grows exponentially over time for both extractors ($r = 1.0, p < 0.001$), where most of the data points lie in the time frame 2002–2007. Assuming an upper bound of $2^{\#F}$, the variability factor (i.e., the ratio of valid to theoretically possible configurations) is $1.9 \cdot 10^{-131}$ (min ≈ 0.0, max $= 9.38 \cdot 10^{-27}$). In addition, the number of configurations largely depends on the architecture and extractor, with 10^{554} configurations per (successfully counted) architecture (min $= 10^{150}$, max $= 10^{876}$) for KConfigReader and 10^{393} (min $= 10^{105}$, max $= 10^{746}$) for KClause.

Discussion Regarding the revision, we observe the exponential growth we expected; regarding the architectures, we see large differences in their number of configurations consistent with their respective number of features. We can also see that the theoretical upper bound of $2^{\#F}$ is not particularly meaningful. Thus, using a #SAT solver is necessary when we are interested in the absolute number of configurations (e.g., to determine by how many orders of magnitude the configuration space has grown) or for more specialized applications [130, 156]. Finally, the number of configurations obtained with KConfigReader is 49.5% longer than with KClause, which is due to the three-valued semantics of

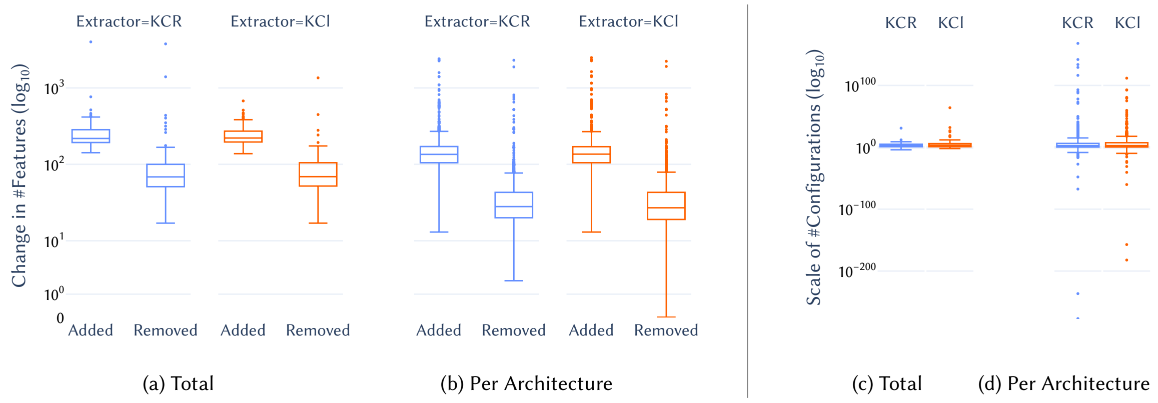


Fig. 5. Change in the number of features (5a–b, change in absolutes) and configurations (5c–d, change in scale) per kernel release.

Table 3. Quartiles of changes shown in Figure 5, rounded to integers.

Quartile	Number of Features								Number of Configurations			
	(a) Total				(b) Per Architecture				(c) Total		(d) Per Architecture	
	KConfigReader		KClause		KConfigReader		KClause		KCR	KCl	KCR	KCl
Q4 (Maximum)	+ 4,005	− 3,772	+ 676	− 1,355	+ 2,406	− 2,303	+ 2,492	− 2,231	$\cdot 10^{31}$	$\cdot 10^{64}$	$\cdot 10^{168}$	$\cdot 10^{112}$
Q3	+ 284	− 99	+ 272	− 104	+ 171	− 43	+ 170	− 43	$\cdot 10^{5}$	$\cdot 10^{6}$	$\cdot 10^{6}$	$\cdot 10^{7}$
Q2 (Median)	+ 218	− 68	+ 221	− 69	+ 135	− 28	+ 136	− 27	$\cdot 10^{2}$	$\cdot 10^{2}$	$\cdot 10^{2}$	$\cdot 10^{2}$
Q1	+ 194	− 52	+ 196	− 52	+ 105	− 20	+ 105	− 19	$\cdot 10^{1}$	$\cdot 10^{1}$	$\cdot 10^{0}$	$\cdot 10^{0}$
Q0 (Minimum)	+ 142	− 17	+ 138	− 17	+ 13	− 3	+ 13	− 1	$\cdot 10^{-4}$	$\cdot 10^{-2}$	$\cdot 10^{-277}$	$\cdot 10^{-182}$

KConfigReader (RQ$_2$). Which result is deemed more accurate depends on the use case; that is, whether non-Boolean features should be included in the number of configurations or not (cf. Section 4).

The number of configurations of Linux grows exponentially over time. It also depends largely on the architecture and used extractor.

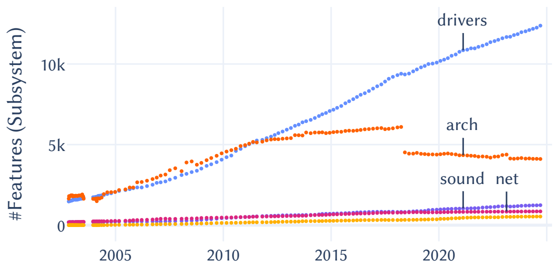
6.3.5 RQ$_{4.1}$: How does the number of features evolve in a typical kernel release? In Figure 5a–b, we show how many features (on a logarithmic scale) have been changed in every kernel release since 2005. Similar to Figure 4a–b, we base our calculations on the set of features F by comparing every two sets of features F_{old} and F_{new} of consecutive kernel releases. Because the underlying set of features F is known (and not only the *number* of features $\#F$), we can distinguish between the added ($F_{new} \setminus F_{old}$) and removed ($F_{old} \setminus F_{new}$) features. We only consider kernel releases since 2005 in order to avoid biased results due to the tighter release cycles from 2002 to 2004 (RQ$_1$). This way, we can infer how many features are added and removed in a typical kernel release, which occurs every two to three months. In addition, we distinguish the extractor and report results for both the total number of features (cf. Figure 5a) and the number of features per architecture (cf. Figure 5b). Finally, we also report the quartiles of the box plots in Table 3.

In addition, as the number of features grows linearly (RQ$_{3.1}$), the question arises how this growth is distributed across different parts of the kernel, also known as its *subsystems* [27, 149] (e.g., hardware drivers, memory management, or scheduling). To investigate the impact of these subsystems on the evolution of the number of features, we show

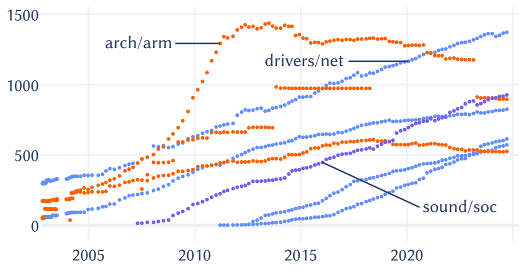
how the number of features develops in several kernel subsystems over the course of time in Figure 6. To this end, we take the union of the set of features F for both extractors and cross-reference each feature with its definition in some KConfig file in the kernel's source tree. We then group features by the directory they are defined in, both in a coarse- and a fine-grained fashion: First, in Figure 6a, we group features by the kernel's top-level directories, each of which corresponds to a major subsystem. Second, in Figure 6b, we group features by subsystems on the second level of the directory hierarchy, which correspond to individual architectures and driver classes. To identify notable drivers of variability in the kernel and keep Figure 6a–b readable, we only show subsystems that changed significantly over the entire period (i.e., where the maximum minus minimum number of features is > 500).

Results As we show in Table 3, a typical kernel release adds 221 features for KClause (218 for KConfigReader) and removes 69 features (68 for KConfigReader) in total (cf. Figure 5a). There are some outliers and differences between the extractors, with up to 4,005 features added and 3,772 features removed in a single release for KConfigReader and up to 676 features added and 1,355 features removed for KClause. When distinguishing architectures (cf. Figure 5b), 136 features (135 for KConfigReader) are added and 27 features (28 for KConfigReader) are removed in the median per architecture. In this case, there are also outliers with up to 2,406 features added and 2,303 features removed for KConfigReader and up to 2,492 features added and 2,231 features removed for KClause. Finally, in neither the total nor per-architecture number of features do we observe that *no* feature is added or removed—there is always *some* change, even for niche architectures.

When we investigate the effect of individual subsystems of the kernel on the evolution of the number of features, we only observe significant changes in five top-level subsystems (cf. Figure 6a): `drivers` (containing device drivers), `arch` (containing architecture-specific code), `sound`, `net` (containing the sound and network abstraction layers), and `lib` (containing library code). Of those four, the `drivers` subsystem is the one that mostly drives the linear growth of the number of features today ($r = 1.0, p < 0.001$). In contrast, the `arch` subsystem, once growing as well, has steadily lost features since the removal of several obsolete architectures in 2018 ($RQ_{3.1}$). The `sound`, `net`, and `lib` subsystems all have slowly accumulated more features ($r \geq 0.99, p < 0.001$), but they still only make up $\approx 13\%$ of the kernel's features today. All other top-level subsystems have not significantly changed over time in terms of the number of features. In Figure 6b, we show eight second-level subsystems whose number of features changed significantly (ordered top to bottom per each individual top-level subsystem color): `arch/arm`, `arch/blackfin`, `arch/mips`, `drivers/net`, `drivers/media`, `drivers/iio`, `drivers/clk`, and `sound/soc`. In particular, we observe that the `arm` architecture steadily gained more architecture-specific features until 2012, where this development stopped, and even reversed in 2015, leading to a steady



(a) Top-Level Subsystems



(b) Second-Level Subsystems

Fig. 6. Number of features in kernel subsystems that significantly changed (> 500 features) over the course of time.

decline instead. Still, ≈ 4% of all features in the Linux kernel today belong specifically to the `arm` architecture. Finally, we find that ≈ 7% of all features in the kernel implement network device drivers in `drivers/net`, a subsystem that is also constantly growing ($r \approx 1.0, p < 0.001$).

Discussion The results are consistent with the number of features' linear growth observed in $RQ_{3.1}$. In addition, we see that every kernel release both adds and (to a lesser degree) removes features such that, for about every three added features, one feature is removed (not considering feature renamings, of which there are few [50, 107, 119]). For the total number of features, the reported outliers and differences between the extractors can be attributed to the unsatisfiable feature-model formulas of KConfigReader in 2011 and the removal of several architectures in 2018 (RQ_1). For the number of features per architecture, the outliers are likely to be due to the in- and exclusion of whole KConfig file trees, which can have a large impact on the number of features.

In terms of subsystems, we find that the most configurable parts of the kernel can all be attributed to the diversity of hardware the kernel is intended to support (in terms of device drivers, architectures, sound, and network facilities). In contrast, the core subsystems of the kernel (e.g., memory management, file systems, and inter-process communication) are much less configurable. In particular, we find that device drivers make up over half of the kernel's features and that they are mostly responsible for the linear growth observed in $RQ_{3.1}$. Furthermore, the gradual reduction of architecture-specific features suggests that kernel developers became aware of the limited reuse potential of such features, thus either removing them fully or making them available in other architectures instead.

A typical release of the Linux kernel adds ≈ 220 features and removes ≈ 70 features in total. In a given architecture, it adds ≈ 130 features and removes ≈ 30 features. This growth is primarily driven by the diversity in supported hardware.

6.3.6 $RQ_{4.2}$: How does the number of configurations evolve in a typical kernel release? Similarly to $RQ_{4.1}$, we show how many configurations have been changed in every kernel release in Figure 5c–d. However, there are some differences to the number of features ($RQ_{4.1}$): First, we cannot distinguish between added and removed configurations, as the underlying set of configurations C is not known (only its cardinality $\#C$). This would require semantic differencing [6, 168], which has not been applied successfully to Linux yet [155]. Second, as many #SAT calls beyond 2007 do not succeed (RQ_1), we consider all kernel releases (not only those since 2005) to get meaningful results. This may slightly bias the results towards tighter release cycles, but cannot be avoided due to the lack of more recent data points. Again, we report the quartiles of the box plots in Table 3.

Finally, as we observe both a linear growth in the number of features ($\#F \sim time$, $RQ_{3.1}$) and an exponential growth in the number of configurations ($time \sim log_{10}(\#C)$, cf. $RQ_{3.2}$), we have reason to suspect that the number of features and the logarithmic number of configurations correlate as well ($\#F \sim log_{10}(\#C)$). To investigate this, we plot the total and per-architecture number of features against the logarithmic number of configurations in Figure 7. We omit the time dimension and include only feature models where #SAT succeeded (i.e., we exclude lower bounds for the total number).

Results As we show in Table 3, a typical kernel release increases the (total and per-architecture) number of configurations by two orders of magnitude (cf. Figure 5c). Similarly to $RQ_{4.1}$, there are outliers and differences between the extractors, with a growth of up to 64 orders of magnitude (31 for KConfigReader). When distinguishing architectures (cf. Figure 5d), larger outliers appear, with a growth of up to 112 orders of magnitude (168 for KConfigReader). Finally, of all analyzed kernel releases, only 4% (7.9% for KConfigReader) decrease the total number of configurations.

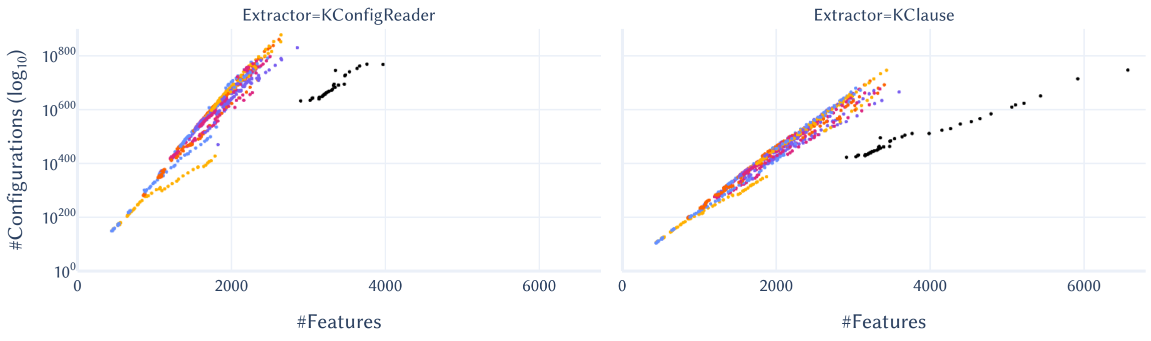


Fig. 7. The number of features correlates with the logarithmic number of configurations (total is black, per-architecture is colored).

As shown in black in Figure 7, we further observe that the total number of features in Linux correlates very strongly with the logarithmic total number of configurations ($r = 0.94, p < 0.001$ for KConfigReader and $r = 0.99, p < 0.001$ for KClause). Similarly, we find very strong and often nearly perfect correlations for both extractors with $r = 1.0$ ($\min = 0.89, \max = 1.0$) and $p < 0.001$ ($\min = 0.001, \max = 0.003$) when we distinguish the individual architectures colored in Figure 7 (only considering those with at least four data points).

Discussion We find that most kernel releases increase the total number of configurations, which is in line with our results for the evolution of the number of features ($RQ_{4.1}$). The outliers are due to the same reasons as for $RQ_{4.1}$, while the larger extractor differences can be attributed to the three-valued semantics of KConfigReader (RQ_2).

Furthermore, we observe that, with $r = 1.0$ in the median, the number of configurations of the Linux kernel is strictly exponential in the number of features. This is consistent with similar observations made in product lines other than Linux [158]. However, the degree to which both metrics correlate is remarkable and has, to the best of our knowledge, not yet been observed in other product lines. Another point worth noting is the difference between the total and per-architecture correlation, which becomes apparent in the black data points lying noticeably below the colored data points in Figure 7. This is because the total number of configurations $\#C$ is a sum of exponential terms, which is dominated by the largest architecture, while the total number of features $\#F$ is a union of sets that grows sub-linearly (i.e., most black data points are obtained by shifting the data points from x86 to the right).

A typical release of the Linux kernel increases the total and per-architecture number of configurations by two orders of magnitude. In addition, the number of configurations of the Linux kernel is exponential in its number of features.

6.3.7 $RQ_{5.1}$: How many features does the Linux kernel have, and how many might it have in the future? As determined in $RQ_{3.1}$, the Linux kernel currently has ≈ 20,000 features (as of September 2024). As for the future, we intend to roughly predict how the kernel's number of features might evolve in the next twenty years. Making the strong assumption that kernel development continues similarly, we suspect that a simple linear regression model is suitable for such a prediction. This is motivated by two of our previous observations: First, the number of features has grown linearly up until now ($RQ_{3.1}$). Second, the number of added features per kernel release has remained relatively stable ($RQ_{4.1}$).

To confirm this suspicion and empirically determine whether a linear regression model can accurately predict the number of features, we use a simplified twofold cross-validation scheme [62]. That is, we train the model on the first

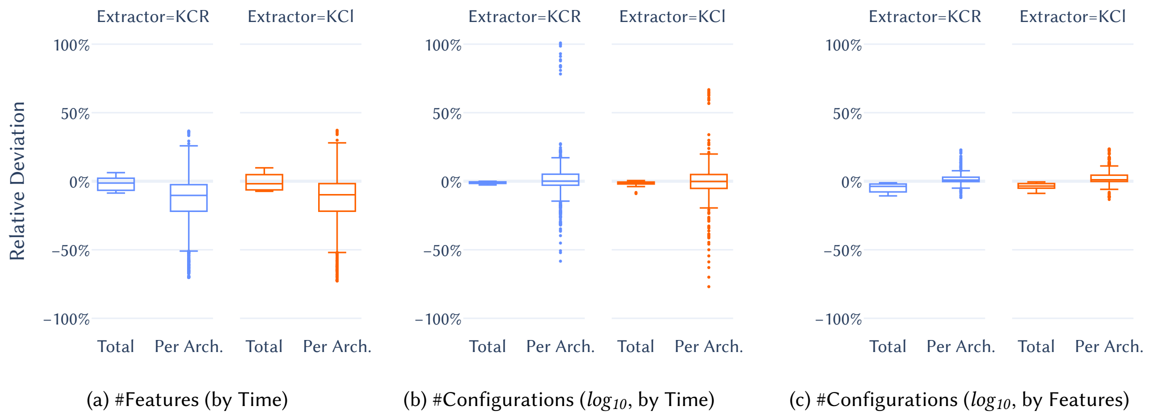


Fig. 8. Relative deviation from the ground truth when predicting kernel configurability metrics by different correlations.

half of a given set of data points, and then we validate it with the second half. For example, we train a model on the total number of features from 2002 to 2013. We then validate this model on the total number of features from 2014 to 2024 and record how much its prediction deviates from the ground truth in this time frame—thus, we effectively evaluate how accurate a ten-year-old prediction would be today. If this accuracy is sufficient, the model is likely to also be suitable for predicting the future, assuming that there are no major changes in the kernel development process.

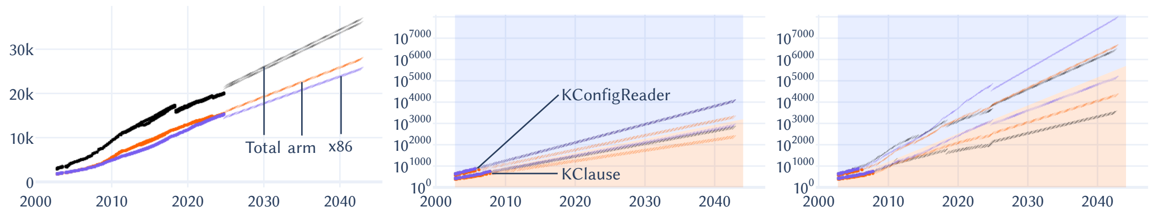
In Figure 8a, we show the relative deviation of a linear regression model for predicting the (total and per-architecture) number of features of Linux. For this prediction, we leverage the correlation between the number of features and the time of the kernel release ($\#F \sim time$, cf. $\text{RQ}_{3.1}$). Thus, the boxes in Figure 8a represent by how much percent our prediction (trained on data points from 2002 to ≈ 2013) deviates from the actual number of features (validated on data points from ≈ 2013 to 2024). We report the median values of this relative deviation in the rightmost column in Table 4. As above, we distinguish between the two extractors KConfigReader and KClause.

In Figure 9a, we show how the kernel's number of features has evolved from 2002 until 2024, and how we predict it to evolve until 2044. To be concise, we only consider the total number of features and the number of features for two influential architectures, `x86` (combined with `i386`) and `arm`. Finally, we report the known and predicted values for every ten years in Table 4 (always referring to the 1st of January of the given year). We also report how many features are added daily, weekly, monthly, and yearly according to the linear regression model.

Results As we show in Figure 8a, the linear regression model predicts the kernel's total number of features with a median relative deviation of −1.3% (min = −8.6%, max = 6.3%) for KConfigReader and −1.8% (min = −7.4%, max = 9.8%) for KClause, and its per-architecture number of features with a median relative deviation of −10.3% (min = −70.2%, max = 36.7%) for KConfigReader and −9.9% (min = −72.7%, max = 37.2%) for KClause.

As for the actual prediction, we show in Figure 9a and Table 4 how we expect the kernel's total number of features to grow from ≈ 19,500 in January 2024 to ≈ 37,000 − 38,000 in 2044, with ≈ 830 − 850 new features per year (treating KClause as lower and KConfigReader as upper bound). We observe similar trends for the architectures `x86` and `arm`, which we report in Table 4.

Discussion Linear regression is generally suitable for predicting the kernel's total number of features, as our ten-year prediction deviates at most by 10% from the ground truth. The per-architecture prediction is less reliable for some



(a) #Features (by Time) (b) #Configurations (log_{10}, by Time) (c) #Configurations (log_{10}, by Features)

Fig. 9. Prediction of configurability metrics for selected kernel architectures (known values are bold, predicted values are thin).

Table 4. Predicted change and values of metrics shown in Figure 9, based on linear regression and rounded to integers.

Configurability Metric		**Predicted Change**				**Predicted Value** (Known Value)					**Deviation**
Architecture	Extractor	Daily	Weekly	Monthly	Yearly	Past		Present	Future		Median
						2004	2014	2024	2034	2044	
Number of Features (by Time, cf. Figure 9a)											(Fig. 8a)
Total	KConfigReader	+ 2	+ 16	+ 69	+ 825	(3,489)	(12,938)	(19,444)	28,710	36,962	− 1.3%
	KClause	+ 2	+ 16	+ 70	+ 845	(3,490)	(13,520)	(19,621)	29,361	37,806	− 1.8%
x86	KConfigReader	+ 2	+ 12	+ 51	+ 612	(2,136)	(7,205)	(14,873)	20,205	26,321	− 6.9%
	KClause	+ 2	+ 12	+ 51	+ 609	(2,134)	(7,311)	(14,766)	20,146	26,235	− 5.6%
arm	KConfigReader	+ 2	+ 13	+ 55	+ 662	(2,108)	(8,479)	(15,036)	21,927	28,550	− 15.0%
	KClause	+ 2	+ 13	+ 55	+ 661	(2,100)	(8,794)	(14,920)	21,907	28,516	− 15.6%
Number of Configurations (by Time, cf. Figure 9b)											(Fig. 8b, log_{10})
Total	KConfigReader	$\cdot 10^{0}$	$\cdot 10^{2}$	$\cdot 10^{7}$	$\cdot 10^{85}$	(10^{728})	$10^{1,587}$	$10^{2,441}$	$10^{3,295}$	$10^{4,149}$	− 1.2%
	KClause	$\cdot 10^{0}$	$\cdot 10^{1}$	$\cdot 10^{5}$	$\cdot 10^{60}$	(10^{484})	$10^{1,090}$	$10^{1,689}$	$10^{2,288}$	$10^{2,887}$	− 1.0%
x86	KConfigReader	$\cdot 10^{0}$	$\cdot 10^{2}$	$\cdot 10^{7}$	$\cdot 10^{85}$	(10^{728})	$10^{1,587}$	$10^{2,441}$	$10^{3,295}$	$10^{4,149}$	− 1.4%
	KClause	$\cdot 10^{0}$	$\cdot 10^{1}$	$\cdot 10^{5}$	$\cdot 10^{62}$	(10^{484})	$10^{1,112}$	$10^{1,732}$	$10^{2,353}$	$10^{2,973}$	− 0.6%
arm	KConfigReader	$\cdot 10^{0}$	$\cdot 10^{1}$	$\cdot 10^{6}$	$\cdot 10^{68}$	(10^{670})	$10^{1,348}$	$10^{2,026}$	$10^{2,704}$	$10^{3,381}$	+ 5.0%
	KClause	$\cdot 10^{0}$	$\cdot 10^{1}$	$\cdot 10^{4}$	$\cdot 10^{49}$	(10^{445})	10^{944}	$10^{1,438}$	$10^{1,933}$	$10^{2,428}$	+ 5.1%
Number of Configurations (by Features, cf. Figure 9c)											(Fig. 8c, log_{10})
Total	KConfigReader	$\cdot 10^{0}$	$\cdot 10^{3}$	$\cdot 10^{12}$	$\cdot 10^{146}$	(10^{728})	$10^{2,385}$	$10^{3,536}$	$10^{5,177}$	$10^{6,638}$	− 3.8%
	KClause	$\cdot 10^{0}$	$\cdot 10^{1}$	$\cdot 10^{6}$	$\cdot 10^{78}$	(10^{484})	$10^{1,390}$	$10^{1,949}$	$10^{2,843}$	$10^{3,618}$	− 3.5%
x86	KConfigReader	$\cdot 10^{1}$	$\cdot 10^{4}$	$\cdot 10^{16}$	$\cdot 10^{188}$	(10^{728})	$10^{2,287}$	$10^{4,647}$	$10^{6,289}$	$10^{8,172}$	− 0.1%
	KClause	$\cdot 10^{0}$	$\cdot 10^{2}$	$\cdot 10^{10}$	$\cdot 10^{122}$	(10^{484})	$10^{1,518}$	$10^{3,008}$	$10^{4,084}$	$10^{5,301}$	0.1%
arm	KConfigReader	$\cdot 10^{0}$	$\cdot 10^{3}$	$\cdot 10^{13}$	$\cdot 10^{154}$	(10^{670})	$10^{2,145}$	$10^{3,673}$	$10^{5,279}$	$10^{6,822}$	+ 1.0%
	KClause	$\cdot 10^{0}$	$\cdot 10^{2}$	$\cdot 10^{8}$	$\cdot 10^{100}$	(10^{445})	$10^{1,455}$	$10^{2,385}$	$10^{3,446}$	$10^{4,449}$	+ 4.6%

outlier architectures, which is probably due to the in- and exclusion of whole KConfig file trees ($RQ_{4.1}$). Usually, we underestimate the number of features, which is likely due to the lower growth in 2002–2013 compared to 2014–2024.

Of course, our future prediction (from 2025 to 2044) assumes that kernel development continues similarly as it has until now. This is a strong assumption, as other scenarios for future evolution are well possible (e.g., changes in development processes, policies, or leadership). Nonetheless, of all the possible scenarios, we consider the linear growth model to be the most conservative assumption (i.e., more speculative assumptions are in scope for future work). Indeed, the linear growth model precisely fulfills our goal in predicting configurability metrics—to understand how the Linux kernel will evolve if no management decisions are made to limit (or accelerate) its growth, and to plan accordingly.

Today, the Linux kernel has ≈ 20,000 features in total. In twenty years, we expect this number to almost double to ≈ 37,000 – 38,000. Typically, ≈ 840 new features are added per year, which corresponds to ≈ 2 new features per day.

6.3.8 RQ$_{5.2}$: How many configurations does the Linux kernel have, and how many might it have in the future? Similarly to RQ$_{5.1}$, we aim to predict the (total and per-architecture) number of configurations of the Linux kernel. However, there are some differences to the number of features (RQ$_{5.1}$): First, the number of configurations in the kernel is not known for any architecture beyond 2013 (RQ$_1$). Thus, our prediction concerns not only the future—it also estimates the past and present number of configurations. Also, this means that we have less data points to validate our prediction, which presumably makes it less reliable. Second, while we rely on linear regression again, we have two correlations that we might leverage for our prediction: the correlation between the logarithmic number of configurations and the time of the kernel release ($log_{10}(\#C) \sim time$, cf. RQ$_{3.2}$), as well as the transitive correlation between the logarithmic number of configurations and the number of features ($\#F \sim log_{10}(\#C)$, cf. RQ$_{4.2}$).

These two correlations give rise to two different prediction techniques, which we describe and evaluate separately. The first technique, predicting configurations *by time*, is analogous to our prediction of the number of features (i.e., we query a linear regression model trained on Figure 4c–d). However, with this technique, we expect limited accuracy due to the lack of ground truth, as we have no data points beyond 2013. Thus, we also consider a second technique, predicting configurations *by features*, which leverages the ground-truth knowledge we have about the past and present number of features. We do so in two steps: First, we predict the number of features in a given year using the linear regression model from RQ$_{5.1}$. If this number is already known (e.g., in the past and present), we omit this prediction and use the known value instead. In a second step, we then predict the number of configurations by using the transitive correlation from RQ$_{4.2}$ (i.e., we query a linear regression model trained on Figure 7). This second technique incorporates the fluctuations in the number of features beyond 2013 into our prediction, which might improve its accuracy compared to the by-time prediction.

Analogous to RQ$_{5.1}$, we show the relative deviation of linear regression models for both prediction techniques in Figure 8b–c. We show in Figure 9b–c how the kernel's number of configurations has evolved from 2002 until 2007, and how we predict it to evolve until 2044. As above, we focus on two selected architectures. In terms of extractors, we show the results for KConfigReader and KClause in the upper and lower half of Figure 9b–c, respectively. Finally, we report the known and predicted values for every ten years in Table 4, as we also did for RQ$_{5.1}$.

Results As we show in Figure 8b, the by-time prediction of the kernel's total number of configurations has a median logarithmic relative deviation of -1.2% ($\min = -2.7\%, \max = 0.0\%$) for KConfigReader and -1.0% ($\min = -9.0\%, \max = 0.5\%$) for KClause; per-architecture it has a deviation of 0.1% ($\min = -58.3\%, \max = 100.9\%$) for KConfigReader and -0.2% ($\min = -76.9\%, \max = 66.8\%$) for KClause. On the other hand, the by-features prediction (cf. Figure 8c) has a deviation of -3.8% ($\min = -10.7\%, \max = -1.1\%$) for KConfigReader and -3.5% ($\min = -8.9\%, \max = 0.5\%$) for KClause; per-architecture it has a deviation of 0.5% ($\min = -11.9\%, \max = 22.8\%$) for KConfigReader and 1.1% ($\min = -13.3\%, \max = 23.6\%$) for KClause.

As for the actual prediction (cf. Figure 9b–c and Table 4), we estimate that, in January 2024, the kernel has $\approx 10^{1{,}689} - 10^{2{,}441}$ configurations by-time and $\approx 10^{1{,}949} - 10^{3{,}536}$ by-features (the lower and upper bound denoting the two- and three-valued semantics of KClause and KConfigReader, respectively, cf. RQ$_{3.2}$). In 2044, we expect this number to grow to $\approx 10^{2{,}887} - 10^{4{,}149}$ by-time and $\approx 10^{3{,}618} - 10^{6{,}638}$ by-features. Thus, we expect a yearly growth by a factor of $\approx 10^{60} - 10^{85}$ by-time and $\approx 10^{78} - 10^{146}$ by-features. For the architectures `x86` and `arm`, we find comparable trends and report them in Table 4. In particular, we predict that the `x86` architecture will have $\approx 10^{2{,}973}$ configurations by-time and $\approx 10^{5{,}301}$ by-features in 2044, where the latter surprisingly even exceeds our prediction for the total number of configurations in the kernel in 2044 (analogous for KConfigReader).

Discussion The by-time linear regression for the number of configurations has low median deviations (cf. Figure 9b), similar to its number-of-features counterpart ($RQ_{5.1}$). However, the former is based on far fewer data points and deviations are measured logarithmically, so we expect predictions to be less reliable than for the number of features. As an alternative, the by-features prediction is more reliable for predicting the per-architecture number of configurations when the number of features is already known (cf. Figure 9c). On the other hand, it slightly underestimates the total number of configurations compared to the by-time prediction.

The limited accuracy of our predictions becomes apparent in Figure 9b–c: That is, we predict the `x86` architecture to have more configurations in twenty years than the total kernel, which is impossible. This can be attributed to the lack of data points for the total number, as well as the by-time prediction's underestimation of the total number of configurations. Nonetheless, our evaluation constitutes the first attempt to approximate the Linux kernel's number of configurations, and our predictions are a reasonable starting point for guiding management decisions regarding kernel development.

Today, the Linux kernel has $\approx 10^{1,600} - 10^{3,600}$ configurations in total. We expect this number to grow by $\approx 10^{1,200} - 10^{2,800}$ in twenty years. Typically, there is a ten- to thousandfold increase per week and by $\approx 10^{60} - 10^{146}$ per year.

6.4 Insights and Recommendations

Based on our results and discussion of the research questions RQ_1–RQ_5, we derive the following insights I_1–I_{11} regarding the configurability of the Linux kernel. Using these, we make recommendations R_1–R_7 for researchers and practitioners. Some of our insights partially confirm and strengthen the results of previous studies on shorter excerpts of the kernel history, which we cite accordingly. However, most of our insights (in particular with regard to the number of configurations) are completely novel. We did not find any insights that contradict previous studies.

6.4.1 Number of Features. Regarding the number of features, we draw the following conclusions from our evaluation.

I_1 Counting features succeeds reliably. The number of features (as per our definition in Section 4) is not necessarily *easy* to compute on a technical level, as it requires several involved steps (like extracting and transforming a feature-model formula to identify and omit dead features, cf. Section 4). However, once a suitable pipeline is set up (e.g., using TORTE [29] and CadiBack [26]), we succeed in counting the number of features for any revision and architecture in a matter of minutes (a few unsatisfiable or unextractable exceptions aside, cf. RQ_1). Thus, the number of features is a robust and reliable metric, if computed correctly.

I_2 It matters how we count features. In RQ_2, we consider several possible definitions of what counts as a feature. We find that the chosen definition significantly impacts the metric. In particular, extracting a feature-model formula is necessary to remove dead features, which significantly affect the final result (confirming [135]). We find that both extractors construct almost the same set of features F, which suggests that our definition of F is suitable and that our results are reliable.

I_3 Each architecture is unique. We find that the number of features varies significantly between architectures ($RQ_{3.1}$). In particular, focusing exclusively on one influential architecture (e.g., `x86`) is not sufficient to analyze the kernel in its entirety, as it misses out on many features (confirming [48, 50, 135]).

I_4 The number of features grows linearly. While both feature additions and removals are common, their net number grows steadily ($RQ_{3.1}$, $RQ_{4.1}$, confirming [107]). This confirms Lehman's laws of increasing complexity and continuing growth [79, 102]. While this growth in itself is not surprising, its almost perfect linearity is, as well as its magnitude: The former is due to the kernel's highly-regulated development process, which is designed to offer no surprises and ensure maintainability [51]. The latter is a testament to both the ever-increasing diversity in supported hardware and intended backward compatibility of the kernel. Given its steady growth, the question arises whether the quality of the Linux kernel can be maintained in the long run, or whether it will decline as suggested by Lehman's laws [102]. Thus, while the growth in features seems effortless, it comes at the price of increased complexity and maintenance effort.

I_5 Hardware diversity drives the growth. The growth in features (I_4) is largely driven by hardware-related subsystems (i.e., device drivers and architectures, cf. $RQ_{4.1}$), while others are relatively stagnant or only grow moderately (confirming [107, 127]). Presumably, many of these device drivers are independent and can be developed and maintained in relative isolation.

I_6 The number of features might double in twenty years. Due to the near-linear growth in features (I_4), the expectation that new hardware will keep driving the growth (I_5), and the extensive ground truth ($RQ_{3.1}$, $RQ_{4.1}$), we can predict the number of features in the future with reasonable confidence ($RQ_{5.1}$). Per this prediction, we expect the number of features to double in twenty years, if no major management decisions are made to limit its growth.

6.4.2 Number of Configurations. We move on to discussing insights derived from the number of configurations.

I_7 Counting configurations does not succeed on recent revisions and architectures. We are able to count the number of configurations for old revisions and less complex architectures in a matter of minutes to hours (RQ_1). However, we fail to do so for recent revisions and more complex architectures (confirming [130, 153]). This is due to the high computational complexity of the underlying #SAT problem, which state-of-the-art solvers are not able to solve yet for the kernel [98, 166]. In contrast to previous attempts at counting the number of configurations of the Linux kernel [130, 153], we are now able to pinpoint when and for which architectures counting configurations becomes impossible, as we are the first to analyze the entire development history of the kernel's feature model.

I_8 It matters how we count configurations. Similar to the number of features (I_2), the number of configurations can be counted in different ways, which impact the metric (RQ_2). The biggest obstacle to giving a precise definition is the presence of non-Boolean features (i.e., `int`, `hex`, and `string`). While these features can be encoded in a formula with bit-blasting [117] or SMT solving [151], both techniques do not scale to the Linux kernel. More critically, these features would drastically skew the metric if we included them [9, 114]. That is, already for as few as 32 Boolean features, a single 4-byte `int` feature would occupy half the configuration space and therefore distort it. Even for the three-valued `tristate` features, it is unclear whether they should be encoded with two-valued (KClause) or three-valued (KConfig-Reader) logic, as it often does not matter whether a feature is compiled statically or as a module. In addition, we expect that the different values of non-Boolean features mostly make minor changes to the compiled kernel [48] (i.e., in terms of changed few bytes), while Boolean features typically in- or exclude whole code blocks (i.e., giving rise to whole new feature interactions [31]). Thus, while the handling of non-Boolean features is an interesting, but distinct research problem, we focus on the Boolean fragment of the kernel in this work and consider both two- and three-valued logic by evaluating two extractors. While our concrete results differ due to the two distinct `tristate` encodings, we observe the exact same trends for both extractors, which suggests that our results are reliable.

Besides non-Boolean features, it is also important to correctly incorporate unconstrained features. Similar to omitting dead features from the number of configurations, this step can easily be overlooked, although it has considerable impact. Thus, by using our definition of the number of configurations, we can obtain the most accurate results yet.

I_9 The number of configurations grows exponentially over time. Analogous to the number of features (I_4), the number of configurations grows steadily in the small time frame that we were able to analyze. However, it does so in an exponential instead of a linear manner ($RQ_{3.2}$). As in other product lines, the number of configurations is also exponential in the number of features [153], this confirms that the Linux kernel is no exception in this regard.

I_{10} The number of features and configurations measure configurability equally well. Upon closer inspection, we find that the number of configurations is even *strictly* exponential in the number of features (i.e., they correlate very strongly to nearly perfectly, cf. $RQ_{4.2}$). This means that both metrics are equally suitable to measure the evolution of the kernel's configurability. This is surprising, as our definition of the number of features ignores most constraints between features, although these constraints are crucial to determine the number of configurations. For example, a new feature in the kernel may either double the number of configurations $\#C$ (if it is unconstrained), or have no effect on $\#C$ (if it is core or dead), or anything in between. Thus, our results show that both developer activities, adding new features and constraining them, are essentially in balance, suggesting that most features are uniformly constrained. In particular, this refutes the (in principle reasonable) hypothesis that the number of configurations is a much more accurate metric for configurability than the number of features. At least for the Linux kernel in 2002–2007, this is not true, possibly due to its strict development process and steady leadership. It is still unknown whether this insight generalizes to other product lines or more recent kernels.

I_{11} The Linux kernel is likely the largest product line. While we cannot count the precise number of configurations for today's kernel (I_7), we can predict it based on either the by-time or by-features correlation ($RQ_{5.2}$). In contrast to the number of features, our prediction is less reliable, as we only have ground truth for the time frame 2002–2007. In addition, our prediction differs based on the chosen correlation and extractor. However, the consensus of all our predictions is that, as of today, the Boolean fragment of the Linux kernel's feature model is likely to have somewhere in between $\approx 10^{1,600} - 10^{3,600}$ configurations. Thus, Linux is likely to surpass even the largest currently known industrial product line in terms of configurability, which has $1.7 \cdot 10^{1,534}$ configurations [153]. The sheer magnitude of this system (and its ensuing complexity) is difficult to put into perspective, as it by far exceeds the number of all humans that ever lived ($1.17 \cdot 10^{11}$ [134]), the estimated number of atoms in the universe ($5.3 \cdot 10^{79}$ [129]),[38] and even the number of different proteins a human cell can possibly produce ($6.8 \cdot 10^{495}$ [29]),[39] which it surpassed about twenty years ago.

6.4.3 Recommendations for Researchers. Based on the insights I_1–I_{11}, we make the following recommendations for researchers, which they might follow when studying variability in the Linux kernel.

R_1 Measure configurability by counting features. The number of features is a robust and reliable metric for the configurability of Linux, as it can be computed in a matter of minutes (I_1) and correlates well with the number of configurations (I_{10}). The common practice in the literature (cf. Table 1), where the number of features is used as a fundamental metric for configurability (and, thus, complexity), is therefore justified for the Linux kernel. This is reassuring, as the alternative (counting the number of configurations with a #SAT solver) is currently too computationally difficult to be applied to configuration spaces as large as that of Linux (I_7).

[38]https://physics.stackexchange.com/q/322880/
[39]https://sites.google.com/view/sources-impossible-machines/

R_2 Count features and configurations correctly. While measuring the configurability of Linux with the number of features is in principle justified (R_1), researchers should take care to count features correctly (e.g., according to our definition or an equally suitable one, cf. I_2). In particular, when analyzing only one architecture of Linux, dead features should be detected and removed by extracting and transforming feature-model formulas. Otherwise, the number of features is not a reliable and comparable metric, as we have illustrated in Table 1. The same applies to counting configurations, where unconstrained features must be incorporated, and the approach for addressing non-Boolean features must be disclosed transparently (I_8). As knowledge about both features and configurations affects most feature-model analyses in some way, incorrect results may propagate downstream and lead to incorrect conclusions about the kernel, which should be avoided.

R_3 Analyze varying revisions and architectures. Researchers should not focus exclusively on one architecture, as the set of features varies significantly between architectures (I_3). In particular, the `x86` architecture, which the literature exclusively focuses on (cf. Table 1), does not cover one fourth of the features. This may, for example, create blind spots in studies on kernel bugs [1] (e.g., missing bugs that only occur on `arm` smartphones). Instead, researchers should strive to analyze all architectures, as we have done in this work, or at least a selection of representative architectures. Similarly, the research community should move away from analyzing only few kernel revisions (like `v2.6.33`). Instead, researchers can rely on our newly-published history of feature models to identify those kernel releases that their analysis scales to (e.g., with binary search). Because Linux is one of the most configurable product lines to date (I_{11}), it can also serve as a benchmark to determine the current frontiers of feature-model analysis in general.

R_4 Further improve reasoning techniques. While the number of features is a good substitute for the number of configurations to assess the evolution of configurability (R_1), it is not sufficient when we are interested in the absolute size of the configuration space (e.g., to compare different product lines) or to address more advanced use cases, such as feature prioritization [156], uniform random sampling [130], or feature-model slicing [4] and differencing [6]. To apply these and other advanced feature-model [18] and product-line [167] analyses to today's Linux kernel, we still need a substantial improvement in state-of-the-art reasoning techniques, such as #SAT solving [153] and knowledge compilation [76, 166]. In addition, we need to develop and improve upon existing compositional reasoning techniques [106, 141] and the underlying theory [45], so we can leverage the division of the kernel into distinct subsystems and divide-and-conquer analyses (I_5). Otherwise, we are bound to keep analyzing twenty-year-old releases, which are irrelevant to practitioners.

R_5 Use the Linux kernel to push analyses to their limits. Building on R_3 and R_4, our data also gives rise to a new metric for assessing the scalability of feature-model analyses, which is based on the kernel's history: That is, while some analyses may scale to fifteen-year-old versions of Linux (cf. Table 1), they may not scale to current releases. By determining the first kernel release where an analysis can no longer be computed, researchers can assess and compare the complexity [98] of various analyses (e.g., based on SAT [18], #SAT [156], algebra [4, 6], or knowledge compilation [76, 157]). Thus, they may identify problematic and promising analysis techniques. In particular, our history of feature models enables researchers to find *some* kernel release they can successfully analyze, instead of omitting the Linux kernel completely from their evaluations due to a lack of scalability.

6.4.4 Recommendations for Practitioners. We also make several recommendations for Linux practitioners, by which we mostly refer to kernel developers and maintainers.

R_6 Use configurability metrics to make decisions. The number of features is a robust and reliable metric for configurability, which practitioners can use to assess variability in the kernel and its evolution (R_1). There are several concrete use cases in which this metric can inform decision-making [156]: First, sudden spikes in the number of features may point to anomalies in the kernel's development, for example indicating a refactoring, reorganization, or removal of obsolete features. By cross-referencing configurability metrics with other metrics (e.g., source lines of code), more complex anomalous patterns can be detected (e.g., if a patch introduces more features than usual for its affected amount of code). By computing this metric for subsystems (I_5), the individual subsystems' lieutenants (i.e., chief developers) can easily and automatically detect such anomalies in their subsystems and take appropriate actions depending on the goals for the next release (e.g., approve, reject, or postpone patches). Second, monitoring the configurability of a subsystem can help determine its complexity [98], maintainability [16], and how prone it may be to variability bugs [1, 113]. By comparing the configurability of subsystems, developers can then identify those that require more attention, and prioritize accordingly. Third, reliable knowledge about current trends (I_4, I_9) and predicted developments (I_6) can be used to scope the extent of new features [139] and, thus, to shape future kernel development.

R_7 Reduce variability where possible. Kernel developers seem to be generally aware of the potential disadvantages of too extensive variability, as evidenced by the removal of obsolete architectures ($RQ_{3.1}$) and the reduction of architecture-specific features ($RQ_{4.1}$). Still, the kernel is growing more configurable at an astonishing pace (I_4, I_9), which further increases the gap between researchers' reasoning methods and practitioners' needs (I_{10}). To keep this gap from widening, we encourage practitioners to continue to actively reduce variability in the kernel where possible. For example, this can be done by deprecating and then phasing out drivers or architectures that are obsolete, overcomplicated, or barely used. Other researchers have made similar recommendations [7, 158], and our own industry partners also aim to reduce variability in the long term. Together with researchers' efforts to further improve reasoning techniques (R_4), reducing the amount of variability will contribute to improving the scalability of feature-model analyses and the kernel's maintainability. In the long run, this will allow kernel developers to apply years of useful product-line research [18, 105, 156, 167] and tool support [5, 64, 110] and leverage their untapped potential.

6.5 Threats to Validity

We discuss potential threats to the validity of our evaluation as categorized by Wohlin et al. [179].

Conclusion Validity Conclusion validity is concerned with issues that affect our ability to draw correct conclusions from our data. We ensure this type of validity by applying appropriate statistical tests. For example, we use the (nonparametric) Wilcoxon signed-rank test instead of the (parametric) paired Student's t-test, as we cannot assume normality, which the latter requires. In addition, we have a large dataset of feature models that we investigate, ensuring sufficient statistical power. Finally, we do not test many hypotheses, thus avoiding the problem of multiple comparisons.

Internal Validity Internal validity is given when our inferences correctly demonstrate causal relationships. To this end, we should mitigate confounding factors and measurement errors. As our evaluation has several stages, each of which runs different tools, there may in principle be errors in our results. We reduce the risk for such errors as follows: First, we manually compare our results with smaller, manual pre-experiments. Second, we use two competitive #SAT solvers to reduce the number of timeouts. Third, we test the sensitivity of our evaluation parameters. To reduce confounding factors (e.g., randomness in time measurements), we run all experiments on the same machine and ensure that it is not used for other tasks. Finally, we provide all our data and tools to allow for transparent reproducibility.

Construct Validity Construct validity is given when our experiment setting actually reflects the construct under study. Crucially, this requires an accurate mapping of KConfig specifications (the configuration language of the Linux kernel) onto the feature-model formulas studied in our evaluation. We explicitly investigate the quality of this mapping in RQ_2 by comparing several feature and configuration candidates. Still, our mapping from KConfig onto propositional formulas might have some weaknesses, which we discuss in the following.

Notably, in measuring the number of configurations, we only consider Boolean and `tristate` features and neither `int`/`hex` nor `string` features. However, both KConfigReader and KClause fully encode the Boolean fragment of the feature model and are able to reflect some transitive dependencies over non-Boolean features (cf. Section 4) [124]. Because of this and the cross-referencing of two independent extractors, our encoding of the Boolean fragment is likely to be accurate. As argued above (I_8), we consider the handling of non-Boolean features in this context to be a separate research problem. Regardless of whether and how non-Boolean features are counted, we do not expect this to affect the conclusions of our evaluation, as we expect such features to be constrained similarly. Finally, an even more precise metric for configurability would be the number of program variants (i.e., possible kernel executables) [169], which would take into account the C code of the kernel. However, this would require considerable tooling and even more complex feature-model formulas [115], so it seems infeasible to compute with state-of-the-art solving techniques.

Moreover, we disregard some edge cases and special functionalities of the KConfig language. First, regarding the number of added and removed features, we do not detect feature renamings, which could lead to an overestimation. However, renamings only impacts $RQ_{4.1}$, and we also consider them to have presumably low impact, as they are not common in the kernel [50, 119]. Second, by ignoring visibility conditions, we do not distinguish between features that are user-selectable (i.e., those with a `prompt`) and those that are not, which are less common [146]. This is reasonable (and commonly done, cf. Table 1), as both types of features are meaningful to researchers and kernel developers—the distinction is more relevant when studying end-user-facing configurators, which is not the focus of this study. Finally, we do not consider the kernel's feature hierarchy [181] (i.e., the feature tree shown in `menuconfig`). While this hierarchy is useful for some analyses, it does not affect configurability metrics, which our evaluation focuses on.

External Validity External validity is concerned with the generalizability of our results to other contexts. The generalizability of our evaluation may in principle be limited due to our choices regarding the source tree, revision, architecture (cf. Section 3), as well as the extractor, transformation, and analysis (cf. Section 4). However, we believe that our choices are either representative (e.g., we evaluate the mainline kernel and two extractors) or even comprehensive (e.g., we evaluate almost all tagged revisions and architectures). Regarding the analysis, we compute the number of features and configurations, and as a byproduct of our transformation, also a consistency and backbone analysis [18]. These analyses constitute fundamental knowledge about a product line and form the basis for advanced analyses [18, 156], so we consider them a reasonable starting point.

It is yet to be determined whether our results can be generalized to other product lines than Linux (e.g., whether the numbers of features and configurations measure configurability equally on other product lines, too). As we focus on the Linux kernel in this paper, we leave this generalization to future studies with broader scopes.

Finally, our future predictions are speculative by nature, as there is no guarantee for continued linear growth in the Linux kernel. However, we critically reflect on this assumption ($RQ_{5.1}$), which is the most conservative among several possible scenarios (e.g., policy changes, plateauing or abandonment of development, or removal of old hardware or architectures). Nonetheless, we believe that investigating and communicating potential future prospects might help foster a constructive discussion on the evolution of the kernel's configurability among researchers and practitioners.

7 Related Work

In the following, we discuss work that is related to our work in this paper.

Variability in the Linux Kernel There are several studies on the evolution of Linux in general [69, 79] as well as on its feature model [50, 51, 86, 96, 107, 125, 127, 135, 146]. We describe selected lines of related work in more detail.

First, Kamali et al. [86] originally observe that the number of features of Linux is reported inconsistently in the literature (cf. Table 1). We extend their collection with several new data points and provide a detailed analysis of the kernel's number of features and configurations, which is not a focus of their work.

Dintzner et al. [50] investigate architecture-specific changes to the feature model over a time frame of four years. Rothberg et al. [135] extend their work, noting that dead features influence architecture-specific features (cf. RQ_2). We build on their insights (e.g., dead features matter), and extend them significantly (e.g., by counting configurations).

As an early precursor to our work, She et al. [146] extract one of the first feature-model formulas of the Linux kernel, and compare it with existing feature models in the S.P.L.O.T. repository [111]. Lotufo et al. [107] complement their work by investigating the syntactic evolution of the kernel's feature model over a time frame of five years.

While all of these studies motivated our work, we go significantly further, as we extract and publish a wide variety of feature-model formulas of Linux (spanning over twenty years and dozens of architectures),[11] compare extractors, evaluate feature and configuration candidates, count configurations, compare configurability metrics, and predict the potential future of the kernel's configurability, none of which has been done before.

KConfig and Feature-Model Analyses Extracting feature models from the Linux kernel is difficult due to the complex semantics of KConfig [59, 61, 124, 145, 150]. This gave rise to several formalization attempts [124, 145, 178] and a wide variety of extraction approaches [21, 59, 61, 71, 88, 123, 161, 178, 181], some of which have been compared to each other qualitatively [54, 123]. To the best of our knowledge, we perform the first detailed empirical comparison of the influential extractors KConfigReader and KClause for two widely applicable feature-model analyses.

Building on its feature model, advanced product-line analyses [93, 167] can be applied to Linux, for example those involving source code [8, 95, 96, 115, 119] or compiled executables [8, 96, 115, 119]. To this end, one must consider the kernel's build-system [47, 65, 120] and preprocessor variability [55, 89, 104, 105, 162]. Here, we focus on the feature model, which is already challenging to analyze on its own and influences all subsequent analyses.

Feature-Model Datasets and Metrics There are numerous generators for synthetic feature models [144, 154, 163], as well as repositories that collect real-world feature models [111, 132, 152]. However, while some of these repositories contain old revisions (like `v2.6.33`) of the Linux kernel's feature model (cf. Table 1), none of them include recent revisions or a consistent and long-running evolution history of the kernel.

In addition to these datasets, various metrics [16, 24, 78, 98, 153] and tools [56, 77] have been proposed to quantify the configurability, maintainability, or complexity of feature models. We are only aware of one systematic evaluation of these metrics on a large feature-model dataset [153]. In particular, the number of configurations as one fundamental metric for configurability is rarely reported at all [99, 113, 153], and never for (old revisions of) the Linux kernel.

8 Conclusion

The Linux kernel is seen as a milestone for the analysis of configurable systems, and its configurability is an equally fundamental measure. It is key to judging and communicating the kernel's complexity and make informed decisions for its future development. Moreover, the kernel's configurability is closely connected to the scalability and accuracy of advanced product-line analyses on the kernel. Thus, configurability (as measured by the number of features and configurations) also aids in improving our understanding of product-line analyses on Linux in general.

In this paper, we have presented a comprehensive, empirical study of the Linux kernel's configurability, which spans its feature model's entire history from 2002 to 2024. We found that the Linux kernel grows more complex and configurable, with ≈ 840 new features and $\approx 10^{60} - 10^{146}$ times more configurations per year. Indeed, today's kernel is likely the largest known product line in terms of configurability, with $\approx 20{,}000$ features and $\approx 10^{1{,}600} - 10^{3{,}600}$ configurations in total. Furthermore, we even expect the number of features to approximately double in twenty years. Regarding scalability, we could count features successfully; however, it is still infeasible to count the configurations of recent revisions of Linux. While this severely limits the practical applicability of #SAT-based analyses for kernel development, we can at least reasonably approximate the number of configurations due to the kernel's steady growth. Regarding accuracy, the number of features and configurations largely depend on how they are measured. In particular, we proposed a novel definition of features that reconciles diverging analysis results. Regarding influence factors, we found that the number of features depends largely on the analyzed feature model (i.e., the revision and architecture of Linux). The number of configurations additionally depends on the used extractor (e.g., KConfigReader or KClause), mainly due to differences in the encoding of `tristate` features.

With our consistent, two-decade-long history of the kernel's feature model, we partly confirmed previous findings, while also gaining new insights. For instance, each architecture induces a unique feature model, so focusing on one architecture is not representative of the entire kernel. Moreover, the kernel's growth is not uniform across all subsystems, as it is mostly driven by hardware diversity. Finally, we found that the number of features is a robust and reliable metric for the configurability of Linux, as it correlates well with the number of configurations. Based on our results, we want to encourage researchers to report relevant parameters when analyzing the Linux kernel, to accurately count its features, and to identify and push the limits of computationally complex analyses on the kernel. We also encourage practitioners to use configurability metrics when making development decisions, and to reduce variability where possible. Thus, we aim to foster a joint effort by practitioners and researchers to improve the scalability and accuracy of product-line analyses on the Linux kernel in the long term.

In future work, we aim to experiment with (de-)composing the feature model of Linux by "zooming" in and out: Zooming in, we may leverage the modularity of subsystems in the kernel to create smaller, compositional feature models. While this will complicate overarching analyses, the individual models will be easier to analyze, which is beneficial for local analyses. Zooming out, we may create an architecture-unifying feature model for Linux. While this is likely to challenge #SAT-based analyses even more, it will also enable architecture-overarching analyses and eliminate the architecture as a threat to validity. We also aim to study the history of the kernel's feature model in more detail (e.g., to learn appropriate solver parameterizations, apply incremental analyses, or compute feature-model differences).

"We talk about world domination, but we'll neither have it nor deserve it until we learn to do better than this. A lot better." — Eric S. Raymond (2006), author of *The Cathedral and the Bazaar* [131], in answer to Linus Torvalds (cf. Section 1)[40]

[40]http://www.catb.org/~esr/writings/cups-horror.html

Acknowledgments

We would like to thank Thomas Leich, Sven Apel, and Paul Maximilian Bittner for their helpful comments and discussions. Also, many thanks to Paul Gazzillo and Nikolaj Bjørner for their support with KClause and Z3, respectively. Finally, we want to thank the anonymous reviewers for their comments and suggestions.

References

[1] Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *Trans. on Software Engineering and Methodology (TOSEM)* 26, 3, Article 10 (2018), 10:1–10:34 pages. doi:10.1145/3149119

[2] Doug Abbott. 2011. *Linux for Embedded and Real-Time Applications.* Elsevier Science Inc. doi:10.1016/B978-0-7506-7932-9.X5022-2

[3] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2010. Comparing Approaches to Implement Feature Model Composition. In *Proc. Europ. Conf. on Modelling Foundations and Applications (ECMFA).* Springer, 3–19.

[4] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2011. Slicing Feature Models. In *Proc. Int'l Conf. on Automated Software Engineering (ASE).* IEEE, 424–427. doi:10.1109/ASE.2011.6100089

[5] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2013. Familiar: A Domain-Specific Language for Large Scale Management of Feature Models. *Science of Computer Programming (SCP)* 78, 6 (2013), 657–681.

[6] Mathieu Acher, Patrick Heymans, Philippe Collet, Clément Quinton, Philippe Lahire, and Philippe Merle. 2012. Feature Model Differences. In *Proc. Int'l Conf. on Advanced Information Systems Engineering (CAiSE).* Springer, 629–645. doi:10.1007/978-3-642-31095-9_41

[7] Mathieu Acher, Luc Lesoil, Georges Aaron Randrianaina, Xhevahire Tërnava, and Olivier Zendra. 2023. A Call for Removing Variability. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, 82–84. doi:10.1145/3571788.3571801

[8] Mathieu Acher, Hugo Martin, Luc Lesoil, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi, Olivier Barais, and Juliana Alves Pereira. 2022. Feature Subset Selection for Learning Huge Configuration Spaces: The Case of Linux Kernel Size. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, 85–96. doi:10.1145/3546932.3546997

[9] Tobias Achterberg, Stefan Heinz, and Thorsten Koch. 2008. Counting Solutions of Integer Programs Using Unrestricted Subtree Detection. In *Proc. Int'l Conf. on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR).* Springer, 278–282.

[10] Sofia Ananieva, Sandra Greiner, Timo Kehrer, Jacob Krüger, Thomas Kühn, Lukas Linsbauer, Sten Grüner, Anne Koziolek, Henrik Lönn, S. Ramesh, and Ralf H. Reussner. 2022. A Conceptual Model for Unifying Variability in Space and Time: Rationale, Validation, and Illustrative Applications. *Empirical Software Engineering (EMSE)* 27, 5 (2022), 101. doi:10.1007/s10664-021-10097-z

[11] Sofia Ananieva, Matthias Kowal, Thomas Thüm, and Ina Schaefer. 2016. Implicit Constraints in Partial Feature Models. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD).* ACM, 18–27. doi:10.1145/3001867.3001870

[12] Nele Andersen, Krzysztof Czarnecki, Steven She, and Andrzej Wąsowski. 2012. Efficient Synthesis of Feature Models. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, 106–115. doi:10.1145/2362536.2362553

[13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines.* Springer. doi:10.1007/978-3-642-37521-7

[14] Sven Apel and Christian Kästner. 2009. An Overview of Feature-Oriented Software Development. *J. Object Technology (JOT)* 8, 5 (2009), 49–84.

[15] Paolo Arcaini, Angelo Gargantini, and Paolo Vavassori. 2017. Automated Repairing of Variability Models. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, 9–18. doi:10.1145/3106195.3106206

[16] Ebrahim Bagheri and Dragan Gasevic. 2011. Assessing the Maintainability of Software Product Line Feature Models Using Structural Metrics. *Software Quality Journal (SQJ)* 19, 3 (2011), 579–612.

[17] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* Springer, 7–20. doi:10.1007/11554844_3

[18] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–708. doi:10.1016/J.IS.2010.01.001

[19] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. 2007. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS).* Technical Report 2007-01, Lero, 129–134.

[20] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What Is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, 16–25. doi:10.1145/2791060.2791108

[21] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Trans. on Software Engineering (TSE)* 39, 12 (2013), 1611–1640. doi:10.1109/TSE.2013.34

[22] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2010. Variability Modeling in the Real: A Perspective From the Operating Systems Domain. In *Proc. Int'l Conf. on Automated Software Engineering (ASE).* ACM, 73–82. doi:10.1145/1858996.1859010

[23] Rüdiger Berlich. 2001. All You Need to Know About... The Early History of Linux, Part 2. *LinuxUser* (2001).

[24] Carla I. M. Bezerra, Rossana M. C. Andrade, and José Maria S. Monteiro. 2014. Measures for Quality Evaluation of Feature Models. In *Proc. Int'l Conf. on Software Reuse (ICSR)*. Springer, 282–297.

[25] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. 2020. *CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020*. Technical Report Report B-2020-1. University of Helsinki, Department of Computer Science.

[26] Armin Biere, Nils Froleyks, and Wenxi Wang. 2023. CadiBack: Extracting Backbones with CaDiCaL. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 271)*, Meena Mahajan and Friedrich Slivovsky (Eds.). Schloss Dagstuhl, 3:1–3:12. doi:10.4230/LIPIcs.SAT.2023.3

[27] Daniel P. Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel: From I/O Ports to Process Management*. O'Reilly Media, Inc.

[28] David Bretthauer. 2001. Open Source Software: A History. Website. Available online at https://opencommons.uconn.edu/libr_pubs/7; visited on December 5, 2023.

[29] Luciano Brocchieri and Samuel Karlin. 2005. Protein Length in Eukaryotic and Prokaryotic Proteomes. *Nucleic Acids Research* 33, 10 (2005), 3390–3400. doi:10.1093/nar/gki615

[30] Hans Kleine Büning and Theodor Lettmann. 1999. *Propositional Logic: Deduction and Algorithms*. Vol. 48. Cambridge University Press.

[31] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. 2003. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks* 41, 1 (2003), 115–141.

[32] Ivan Do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana De Almeida. 2014. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *J. Information and Software Technology (IST)* 56, 10 (2014), 1183–1199. doi:10.1016/j.infsof.2014.04.002

[33] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. 2013. A Scalable Approximate Model Counter. In *Proc. Int'l Conf. on Principles and Practice of Constraint Programming (CP)*, Christian Schulte (Ed.). Springer, 200–216.

[34] Michael Elizabeth Chastain. 2023. Linux Kernel Makefiles. Website: https://www.kernel.org/doc/html/latest/kbuild/makefiles.html. Accessed: 2023-07-10.

[35] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-Francois Raskin. 2013. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Trans. on Software Engineering (TSE)* 39, 8 (2013), 1069–1089. doi:10.1109/TSE.2012.86

[36] Paul C Clements, John D McGregor, and Sholom G Cohen. 2005. *The Structured Intuitive Model for Product Line Economics (SIMPLE)*. Technical Report. Carnegie Mellon University.

[37] The Kernel Development Community. 2018. KConfig Language. Website: https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html. Accessed: 2024-01-30.

[38] Jonathan Corbet, Greg Kroah-Hartman, and Amanda McPherson. 2016. Linux Kernel Development: How Fast It Is Going, Who Is Doing It, What They Are Doing, And Who Is Sponsoring It. Website. Available online at https://www.linuxfoundation.jp/events/2016/08/linux-kernel-development-2016/; visited on June 29, 2024.

[39] Corbet, Jonathan. 2008. A Guide to the Kernel Development Process. Website. Available online at https://www.kernel.org/doc/html/latest/process/development-process.html; visited on December 5, 2023.

[40] Kevin Crowston, Kangning Wei, James Howison, and Andrea Wiggins. 2008. Free/Libre Open-Source Software Development: What We Know and What We Do Not Know. *ACM Computing Surveys (CSUR)* 44, 2, Article 7 (2008), 35 pages. doi:10.1145/2089125.2089127

[41] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. *Software Process: Improvement and Practice* 10, 2 (2005), 143–169.

[42] Krzysztof Czarnecki and Chang Hwan Peter Kim. 2005. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In *Proc. Int'l Workshop on Software Factories (SF)*. 16–20.

[43] Krzysztof Czarnecki and Krzysztof Pietroszek. 2006. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE)*. ACM, 211–220.

[44] Krzysztof Czarnecki and Andrzej Wąsowski. 2007. Feature Diagrams and Logics: There and Back Again. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. IEEE, 23–34.

[45] Ferruccio Damiani, Michael Lienhardt, and Luca Paolini. 2020. On Two Characterizations of Feature Models. In *Proc. Int'l Colloquium on Theoretical Aspects of Computing (ICTAC) (Lecture Notes in Computer Science (LNCS))*, Violet Ka I. Pun, Volker Stolz, and Adenilso Simao (Eds.). Springer, 103–122. doi:10.1007/978-3-030-64276-1_6

[46] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proc. Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 337–340.

[47] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. A Robust Approach for Variability Extraction From the Linux Build System. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 21–30. doi:10.1145/2362536.2362544

[48] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. Understanding Linux Feature Distribution. In *Proc. of Workshop on Modularity in Systems Software (MISS)*. ACM, 15–20.

[49] Jeff Dike. 2001. User-Mode Linux. In *Linux Showcase & Conference*. USENIX Association, 2.

[50] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2017. Analysing the Linux Kernel Feature Model Changes Using FMDiff. *Software and Systems Modeling (SoSyM)* 16 (2017), 55–76.

[51] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2018. FEVER: An Approach to Analyze Feature-Oriented Changes and Artefact Co-Evolution in Highly Configurable Systems. *Empirical Software Engineering (EMSE)* 23, 2 (2018), 905–952. doi:10.1007/s10664-017-9557-6

[52] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature Location in Source Code: A Taxonomy and Survey. *J. Software: Evolution and Process* 25, 1 (2013), 53–95. doi:10.1002/smr.567

[53] Niklas Eén and Niklas Sörensson. 2004. An Extensible SAT-solver. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT).* Springer, 502–518.

[54] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2015. Analysing the KConfig Semantics and its Analysis Tools. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE).* ACM, 45–54. doi:10.1145/2814204.2814222

[55] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2017. An Empirical Study of Configuration Mismatches in Linux. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, 19–28. doi:10.1145/3106195.3106208

[56] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2019. MetricHaven: More Than 23,000 Metrics for Measuring Quality Attributes of Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, 25–28. doi:10.1145/3307630.3342384

[57] Kevin Feichtinger, Johann Stöbich, Dario Romano, and Rick Rabiser. 2021. TRAVART: An Approach for Transforming Variability Models. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, Article 8, 10 pages. doi:10.1145/3442391.3442400

[58] Wolfram Fenske, Jens Meinicke, Sandro Schulze, Steffen Schulze, and Gunter Saake. 2017. Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line. In *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 316–326. doi:10.1109/SANER.2017.7884632

[59] David Fernandez-Amoros, Ruben Heradio, Christoph Mayr-Dorn, and Alexander Egyed. 2019. A KConfig Translation to Logic With One-Way Validation System. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, 303–308. doi:10.1145/3336294.3336313

[60] Johannes K. Fichte, Markus Hecher, and Florim Hamiti. 2021. The Model Counting Competition 2020. *ACM J. of Experimental Algorithmics (JEA)* 26, Article 13 (2021), 26 pages. doi:10.1145/3459080

[61] Patrick Franz, Thorsten Berger, Ibrahim Fayaz, Sarah Nadi, and Evgeny Groshev. 2021. ConfigFix: Interactive Configuration Conflict Resolution for the Linux Kernel. In *Proc. Int'l Conf. on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* IEEE, 91–100. doi:10.1109/ICSE-SEIP52600.2021.00018

[62] Tadayoshi Fushiki. 2011. Estimation of Prediction Error by Using K-fold Cross-Validation. *Statistics and Computing* 21 (2011), 137–146.

[63] José A Galindo, Mathieu Acher, Juan Manuel Tirado, Cristian Vidal, Benoit Baudry, and David Benavides. 2016. Exploiting the Enumeration of All Feature Model Configurations: A New Perspective With Distributed Computing. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, 74–78.

[64] José A. Galindo, José Miguel Horcas, Alexander Felfernig, David Fernández-Amorós, and David Benavides. 2023. FLAMA: A Collaborative Effort to Build a New Framework for the Automated Analysis of Feature Models. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, 16–19. doi:10.1145/3579028.3609008

[65] Paul Gazzillo. 2017. Kmax: Finding All Configurations of Kbuild Makefiles Statically. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE).* ACM, 279–290. doi:10.1145/3106237.3106283

[66] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI).* ACM, 323–334. doi:10.1145/2254064.2254103

[67] GCC Development Team. 2025. The C Preprocessor. Website: http://gcc.gnu.org/onlinedocs/cpp/index.html. Accessed: 2025-02-17.

[68] Daniel M. German and Jesús M. González-Barahona. 2009. An Empirical Study of the Reuse of Software Licensed under the GNU General Public License. In *IFIP Int'l Conf. on Open Source Systems.* Springer, 185–198.

[69] Michael Godfrey and Qiang Tu. 2000. Evolution in Open Source Software: A Case Study. In *Proc. Int'l Conf. on Software Maintenance (ICSM).* 131–142. doi:10.1109/ICSM.2000.883030

[70] Édouard Guégain, Clément Quinton, and Romain Rouvoy. 2021. On Reducing the Energy Consumption of Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, 89–99. doi:10.1145/3461001.3471142

[71] Jude Gyimah, Jan Sollmann, Ole Schuerks, Patrick Franz, and Thorsten Berger. 2025. A Demo of ConfigFix: Semantic Abstraction of Kconfig, SAT-based Configuration, and DIMACS Export. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM. To appear.

[72] Tarik Hadzic, Sathiamoorthy Subbarayan, Rune M. Jensen, Henrik R. Andersen, Jesper Møller, and Henrik Hulgaard. 2004. Fast Backtrack-Free Product Configuration using a Precompiled Solution Space Representation. In *Proc. Int'l Conf. on Economic, Technical and Organisational Aspects of Product Configuration Systems.* Gamez Publishing, 131–138.

[73] Yvonne Heimowski. 2022. *Simplifying Feature Models for Better Scalability of #SAT Solvers.* Bachelor's Thesis. University of Ulm.

[74] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. 2014. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *IEEE Trans. on Software Engineering (TSE)* 40, 7 (2014), 650–670. doi:10.1109/TSE.2014.2327020

[75] Marc Hentze, Tobias Pett, Thomas Thüm, and Ina Schaefer. 2021. Hyper Explanations for Feature-Model Defect Analysis. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, Article 14, 9 pages. doi:10.1145/3442391.3442406

[76] Tobias Heß, Chico Sundermann, and Thomas Thüm. 2021. On the Scalability of Building Binary Decision Diagrams for Current Feature Models. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, 131–135. doi:10.1145/3461001.3474452

[77] José M. Horcas, José A. Galindo, Lidia Fuentes, and David Benavides. 2024. FM Fact Label. *Science of Computer Programming (SCP)* (2024), 103214. doi:10.1016/j.scico.2024.103214

[78] José M. Horcas, José A. Galindo, Mónica Pinto, Lidia Fuentes, and David Benavides. 2022. FM Fact Label: A Configurable and Interactive Visualization of Feature Model Characterizations. In *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR) (Proc. Int'l Systems and Software Product Line Conf. (SPLC))*. ACM, 42–45. doi:10.1145/3503229.3547025

[79] Ayelet Israeli and Dror G. Feitelson. 2010. The Linux Kernel as a Case Study in Software Evolution. *J. Systems and Software (JSS)* 83, 3 (2010), 485–501.

[80] Pooyan Jamshidi, Miguel Velez, Christian Kästner, Norbert Siegmund, and Prasad Kawthekar. 2017. Transfer Learning for Improving Model Predictions in Highly Configurable Software. In *Proc. Int'l Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 31–41. doi:10.1109/SEAMS.2017.11

[81] Mikoláš Janota. 2008. Do SAT Solvers Make Good Configurators?. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, Vol. 2. University of Limerick, Lero, 191–195.

[82] Mikoláš Janota, Inês Lynce, and Joao Marques-Silva. 2015. Algorithms for Computing Backbones of Propositional Formulae. *AI Communications* 28, 2 (2015), 161–177.

[83] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. 2012. The International SAT Solver Competitions. *AI Magazine* 33, 1 (2012), 89–92. doi:10.1609/aimag.v33i1.2395

[84] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An Algorithm for Generating T-Wise Covering Arrays From Large Feature Models. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 46–55. doi:10.1145/2362536.2362547

[85] David S Johnson. 1992. The NP-Completeness Column: An Ongoing Guide. *J. Algorithms* 13, 3 (1992), 502–524.

[86] Seiede Reyhane Kamali, Shirin Kasaei, and Roberto E. Lopez-Herrejon. 2019. Answering the Call of the Wild? Thoughts on the Elusive Quest for Ecological Validity in Variability Modeling. In *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*. ACM, 143–150. doi:10.1145/3307630.3342400

[87] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Software Engineering Institute.

[88] Christian Kästner. 2017. *Differential Testing for Variational Analyses: Experience From Developing KConfigReader*. Technical Report arXiv:1706.09357. Cornell University Library.

[89] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 805–824. doi:10.1145/2048066.2048128

[90] Holger M. Kienle and Hausi A. Müller. 2010. Rigi—An Environment for Software Reverse Engineering, Exploration, Visualization, And Redocumentation. *Science of Computer Programming (SCP)* 75, 4 (2010), 247–263. doi:10.1016/j.scico.2009.10.007

[91] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is There a Mismatch Between Real-World Feature Models and Product-Line Research?. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 291–302. doi:10.1145/3106237.3106252

[92] Sebastian Krieter, Rahel Arens, Michael Nieke, Chico Sundermann, Tobias Heß, Thomas Thüm, and Christoph Seidl. 2021. Incremental Construction of Modal Implication Graphs for Evolving Feature Models. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 64–74. doi:10.1145/3461001.3471148

[93] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Sebastian Ruland, Malte Lochau, Gunter Saake, and Thomas Leich. 2022. *T-Wise Presence Condition Coverage and Sampling for Configurable Systems*. Technical Report arXiv:2205.15180. Cornell University Library. doi:10.48550/arXiv.2205.15180

[94] Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. 2020. YASA: Yet Another Sampling Algorithm. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 4, 10 pages. doi:10.1145/3377024.3377042

[95] Christian Kröher, Sascha El-Sharkawy, and Klaus Schmid. 2018. KernelHaven: An Experimentation Workbench for Analyzing Software Product Lines. In *Companion Int'l Conf. on Software Engineering (ICSEC)*. ACM, 73–76. doi:10.1145/3183440.3183480

[96] Christian Kröher, Lea Gerling, and Klaus Schmid. 2023. Comparing the Intensity of Variability Changes in Software Product Line Evolution. *J. Systems and Software (JSS)* 203 (2023), 111737. doi:10.1016/j.jss.2023.111737

[97] Andreas Kübler, Christoph Zengler, and Wolfgang Küchlin. 2010. Model Counting in Product Configuration. In *Proc. Int'l Workshop on Logics for Component Configuration (LoCoCo)*. Open Publishing Association, 44–53. doi:10.4204/EPTCS.29.5

[98] Elias Kuiter, Tobias Heß, Chico Sundermann, Sebastian Krieter, Thomas Thüm, and Gunter Saake. 2024. How Easy Is SAT-Based Analysis of a Feature Model?. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 149–151. doi:10.1145/3634713.3634733

[99] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2022. Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 110:1–110:13. doi:10.1145/3551349.3556938

[100] Elias Kuiter, Thomas Thüm, and Timo Kehrer. 2025. Teach Variability! A Modern University Course on Software Product Lines. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM. To appear.

[101] Jihyun Lee, Sungwon Kang, and Danhyung Lee. 2012. A Survey on Software Product Line Testing. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 31–40. doi:10.1145/2362536.2362545

[102] Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry, and Wladyslaw M Turski. 1997. Metrics and Laws of Software Evolution–The Nineties View. In *Proc. Int'l Software Metrics Symposium (METRICS)*. IEEE, 20–32.

[103] Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. 2015. SAT-Based Analysis of Large Real-World Feature Models Is Easy. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 91–100.

[104] Jörg Liebig, Christian Kästner, and Sven Apel. 2011. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Proc. Int'l Conf. on Aspect-Oriented Software Development (AOSD)*. ACM, 191–202. doi:10.1145/1960275.1960299

[105] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 81–91. doi:10.1145/2491411.2491437

[106] Michael Lienhardt, Ferruccio Damiani, Einer Broch Johnsen, and Jacopo Mauro. 2020. Lazy Product Discovery in Huge Configuration Spaces. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 1509–1521. doi:10.1145/3377811.3380372

[107] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2010. Evolution of the Linux Kernel Variability Model. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 136–150.

[108] Gabriele Masina, Giuseppe Spallitta, and Roberto Sebastiani. 2023. On CNF Conversion for Disjoint SAT Enumeration. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 271)*, Meena Mahajan and Friedrich Slivovsky (Eds.). Schloss Dagstuhl, 15:1–15:16. doi:10.4230/LIPIcs.SAT.2023.15

[109] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 643–654. doi:10.1145/2884781.2884793

[110] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability With FeatureIDE*. Springer. doi:10.1007/978-3-319-61443-4

[111] Marcílio Mendonça, Moises Branco, and Donald Cowan. 2009. S.P.L.O.T.: Software Product Lines Online Tools. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 761–762.

[112] Marcílio Mendonça, Andrzej Wąsowski, and Krzysztof Czarnecki. 2009. SAT-Based Analysis of Feature Models Is Easy. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Software Engineering Institute, 231–240.

[113] Austin Mordahl, Jeho Oh, Ugur Koc, Shiyi Wei, and Paul Gazzillo. 2019. An Empirical Study of Real-World Variability Bugs Detected by Variability-Oblivious Tools. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 50–61. doi:10.1145/3338906.3338967

[114] António Morgado, Paulo Matos, Vasco Manquinho, and João Marques-Silva. 2006. Counting Models in Integer Domains. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 410–423.

[115] Johann Mortara and Philippe Collet. 2021. Capturing the Diversity of Analyses on the Linux Kernel Variability. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 160–171. doi:10.1145/3461001.3471151

[116] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proc. Anual Conf. on Design Automation (DAC)*. ACM, 530–535.

[117] Daniel-Jesus Munoz, Mónica Pinto, Lidia Fuentes, and Don Batory. 2023. Transforming Numerical Feature Models into Propositional Formulas and the Universal Variability Language. *J. Systems and Software (JSS)* 204, Article 111770 (2023). doi:10.1016/j.jss.2023.111770

[118] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Trans. on Software Engineering (TSE)* 41, 8 (2015), 820–841. doi:10.1109/TSE.2015.2415793

[119] Sarah Nadi, Christian Dietrich, Reinhard Tartler, Richard C. Holt, and Daniel Lohmann. 2013. Linux Variability Anomalies: What Causes Them and How Do They Get Fixed?. In *Proc. Working Conf. on Mining Software Repositories (MSR)*. IEEE, 111–120. doi:10.1109/MSR.2013.6624017

[120] Sarah Nadi and Ric Holt. 2014. The Linux Kernel: A Case Study of Build System Variability. *J. Software: Evolution and Process* 26, 8 (2014), 730–746. doi:10.1002/smr.1595

[121] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2014. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 907–918. doi:10.1145/2568225.2568300

[122] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. 2018. Anomaly Analyses for Feature-Model Evolution. In *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 188–201. doi:10.1145/3278122.3278123

[123] Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. 2019. *Uniform Sampling From Kconfig Feature Models*. Technical Report TR-19-02. University of Texas at Austin, Department of Computer Science.

[124] Jeho Oh, Necip Fazıl Yıldıran, Julian Braha, and Paul Gazzillo. 2021. Finding Broken Linux Configuration Specifications by Statically Analyzing the Kconfig Language. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 893–905. doi:10.1145/3468264.3468578

[125] Leonardo Passos and Krzysztof Czarnecki. 2014. A Dataset of Feature Additions and Feature Removals From the Linux Kernel. In *Proc. Working Conf. on Mining Software Repositories (MSR)*. ACM, 376–379.

[126] Leonardo Passos, Marko Novakovic, Yingfei Xiong, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2011. A Study of Non-Boolean Constraints in Variability Models of an Embedded Operating System. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD) (Proc. Int'l Systems and Software Product Line Conf. (SPLC))*. ACM, Article 2. doi:10.1145/2019136.2019139

[127] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wąsowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. 2016. Coevolution of Variability Models and Related Software Artifacts. *Empirical Software Engineering (EMSE)* 21, 4 (2016). doi:10.1007/s10664-015-9364-x

[128] David A Plaisted and Steven Greenbaum. 1986. A Structure-Preserving Clause Form Translation. *J. Symbolic Computation* 2, 3 (1986), 293–304.

[129] Planck Collaboration. 2016. Planck 2015 Results. XIII. Cosmological Parameters. *Astronomy & Astrophysics* 594 (2016), A13. doi:10.1051/0004-6361/201525830

[130] Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. 2019. Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?. In *Proc. Int'l Conf. on Software Testing, Verification and Validation (ICST)*. IEEE, 240–251. doi:10.1109/ICST.2019.00032

[131] Eric Raymond. 1999. The Cathedral and the Bazaar. *Knowledge, Technology & Policy* 12, 3 (1999), 23–49.

[132] David Romero-Organvídez, José A Galindo, Chico Sundermann, José-Miguel Horcas, and David Benavides. 2024. UVLHub: A Feature Model Data Repository Using UVL and Open Science Principles. *J. Systems and Software (JSS)* (2024). To appear.

[133] David Romero-Organvídez, Pablo Neira, José A. Galindo, and David Benavides. 2024. Kconfig Metamodel: A First Approach. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 55–60. doi:10.1145/3646548.3676548

[134] Max Roser. 2022. The Future Is Vast - What Does This Mean for Our Own Life? Website: https://ourworldindata.org/the-future-is-vast.

[135] Valentin Rothberg, Nicolas Dintzner, Andreas Ziegler, and Daniel Lohmann. 2016. Feature Models in Linux: From Symbols to Semantics. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 65–72. doi:10.1145/2866614.2866624

[136] Amit Kumar Saha. 2008. A Developer's First Look At Android. *Linux for You* (2008).

[137] Peter H Salus. 1994. *A Quarter Century of UNIX*. ACM/Addison-Wesley.

[138] Abdel Salam Sayyad, Joseph Ingram, Tim Menzies, and Hany Ammar. 2013. Scalable Product Line Configuration: A Straw to Break the Camel's Back. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. IEEE, 465–474. doi:10.1109/ASE.2013.6693104

[139] Klaus Schmid. 2002. A Comprehensive Product Line Scoping Approach and Its Validation. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 593–603.

[140] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. 2006. Feature Diagrams: A Survey and a Formal Semantics. In *Proc. Int'l Conf. on Requirements Engineering (RE)*. IEEE, 136–145. doi:10.1109/RE.2006.23

[141] Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. 2016. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 667–678. doi:10.1145/2884781.2884823

[142] Alexander Schultheiß, Paul Maximilian Bittner, Sandra Greiner, and Timo Kehrer. 2023. Benchmark Generation With VEVOS: A Coverage Analysis of Evolution Scenarios in Variant-Rich Systems. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 13–22. doi:10.1145/3571788.3571793

[143] Sergio Segura. 2008. Automated Analysis of Feature Models Using Atomic Sets. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*, Vol. 2. IEEE, 201–207.

[144] Sergio Segura, José A. Galindo, David Benavides, José A. Parejo, and Antonio Ruiz-Cortés. 2012. BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 63–71. doi:10.1145/2110147.2110155

[145] Steven She and Thorsten Berger. 2010. *Formal Semantics of the Kconfig Language*. Technical Report. University of Waterloo.

[146] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. 2010. The Variability Model of the Linux Kernel. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. 45–51.

[147] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. 2011. Reverse Engineering Feature Models. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 461–470. doi:10.1145/1985793.1985856

[148] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. 2012. SPL Conqueror: Toward Optimization of Non-functional Properties in Software Product Lines. *Software Quality Journal (SQJ)* 20, 3-4 (2012), 487–517. doi:10.1007/s11219-011-9152-9

[149] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. 2007. Is the Linux Kernel a Software Product Line?. In *Proc. Int'l Workshop on Open Source Software and Product Lines (OSSPL)*. IEEE, 9–12.

[150] Julio Sincero and Wolfgang Schröder-Preikschat. 2008. The Linux Kernel Configurator as a Feature Modeling Tool. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. University of Limerick, Lero, 257–260.

[151] Joshua Sprey, Chico Sundermann, Sebastian Krieter, Michael Nieke, Jacopo Mauro, Thomas Thüm, and Ina Schaefer. 2020. SMT-Based Variability Analyses in FeatureIDE. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 6, 9 pages. doi:10.1145/3377024.3377036

[152] Chico Sundermann, Vincenzo Francesco Brancaccio, Elias Kuiter, Sebastian Krieter, Tobias Heß, and Thomas Thüm. 2024. Collecting Feature Models from the Literature: A Comprehensive Dataset for Benchmarking. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 54–65. doi:10.1145/3646548.3672590

[153] Chico Sundermann, Tobias Heß, Michael Nieke, Paul Maximilian Bittner, Jeffrey M. Young, Thomas Thüm, and Ina Schaefer. 2023. Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces. *Empirical Software Engineering (EMSE)* 28, 2 (2023), 38. doi:10.1007/s10664-022-10265-9

[154] Chico Sundermann, Tobias Heß, Rahel Sundermann, Elias Kuiter, Sebastian Krieter, and Thomas Thüm. 2024. Generating Feature Models with UVL's Full Expressiveness. In *Proc. Int'l Workshop on Languages for Modelling Variability (MODEVAR)*. ACM, 61–65. doi:10.1145/3646548.3676602

[155] Chico Sundermann, Elias Kuiter, Tobias Heß, Heiko Raab, Sebastian Krieter, and Thomas Thüm. 2024. On the Benefits of Knowledge Compilation for Feature-Model Analyses. *Annals of Mathematics and Artificial Intelligence (AMAI)* 92, 5 (2024), 1013–1050. doi:10.1007/s10472-023-09906-6

[156] Chico Sundermann, Michael Nieke, Paul Maximilian Bittner, Tobias Heß, Thomas Thüm, and Ina Schaefer. 2021. Applications of #SAT Solvers on Feature Models. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, Article 12, 10 pages. doi:10.1145/3442391.3442404

[157] Chico Sundermann, Heiko Raab, Tobias Heß, Thomas Thüm, and Ina Schaefer. 2024. Reusing d-DNNFs for Efficient Feature-Model Counting. *Trans. on Software Engineering and Methodology (TOSEM)* 33, 8, Article 208 (2024), 32 pages. doi:10.1145/3680465

[158] Chico Sundermann, Thomas Thüm, and Ina Schaefer. 2020. Evaluating #SAT Solvers on Industrial Feature Models. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, Article 3, 9 pages. doi:10.1145/3377024.3377025

[159] Andrew S. Tanenbaum. 1992. LINUX is Obsolete. Usenet. Available online at https://groups.google.com/g/comp.os.minix/c/wlhw16QWltI/m/XdksCA1TR_QJ2; visited on December 5, 2023.

[160] Andrew S. Tanenbaum and Herbert Bos. 2014. *Modern Operating Systems* (4 ed.). Pearson Education.

[161] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *Proc. Europ. Conf. on Computer Systems (EuroSys).* ACM, 47–60. doi:10.1145/1966445.1966451

[162] Reinhard Tartler, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2009. Dead or Alive: Finding Zombie Features in the Linux Kernel. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD).* ACM, 81–86.

[163] SPLOT Development Team. 2020. SPLOT FM Generator. Website: http://52.32.1.180:8080/SPLOT/splot_fm_generator.html. Accessed: 2020-08-23.

[164] Sahil Thaker, Don Batory, David Kitchin, and William Cook. 2007. Safe Composition of Product Lines. In *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE).* ACM, 95–104.

[165] Keir Thomas. 2006. *Beginning Ubuntu Linux: From Novice to Professional.* Apress.

[166] Thomas Thüm. 2020. A BDD for Linux? The Knowledge Compilation Challenge for Variability. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, Article 16, 6 pages. doi:10.1145/3382025.3414943

[167] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys (CSUR)* 47, 1 (2014), 6:1–6:45. doi:10.1145/2580950

[168] Thomas Thüm, Don Batory, and Christian Kästner. 2009. Reasoning About Edits to Feature Models. In *Proc. Int'l Conf. on Software Engineering (ICSE).* IEEE, 254–264. doi:10.1109/ICSE.2009.5070526

[169] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract Features in Feature Modeling. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* IEEE, 191–200. doi:10.1109/SPLC.2011.53

[170] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. 2019. Towards Efficient Analysis of Variation in Time and Space. In *Proc. Int'l Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution).* ACM, 57–64. doi:10.1145/3307630.3342414

[171] Marc Thurley. 2006. sharpSAT - Counting Models With Advanced Component Caching and Implicit BCP. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT).* Springer, 424–429.

[172] Linus Torvalds. 1999. The Linux Edge. *Comm. ACM* 42, 4 (1999), 38–39.

[173] Grigori S. Tseitin. 1983. *On the Complexity of Derivation in Propositional Calculus.* Springer, 466–483. doi:10.1007/978-3-642-81955-1_28

[174] Leslie G Valiant. 1979. The Complexity of Computing the Permanent. *Theoretical Computer Science* 8, 2 (1979), 189–201.

[175] Arie van Deursen and Paul Klint. 2002. Domain-Specific Language Design Requires Feature Descriptions. *Computing and Information Technology* 10, 1 (2002), 1–17.

[176] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC).* ACM, 1–13. doi:10.1145/3233027.3233035

[177] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-Condition Simplification in Highly Configurable Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE).* IEEE, 178–188.

[178] Martin Walch, Rouven Walter, and Wolfgang Küchlin. 2015. Formal Analysis of the Linux Kernel Configuration With SAT Solving. In *Proc. Configuration Workshop (ConfWS).*

[179] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. 2012. *Experimentation in Software Engineering.* Springer. doi:10.1007/978-3-642-29044-2

[180] Yinxing Xue, Jinghui Zhong, Tian Huat Tan, Yang Liu, Wentong Cai, Manman Chen, and Jun Sun. 2016. IBED: Combining IBEA and DE for Optimal Feature Selection in Software Product Line Engineering. *Applied Soft Computing* 49 (2016), 1215–1231. doi:10.1016/j.asoc.2016.07.040

[181] Kaan Berk Yaman, Jan Willem Wittler, and Christopher Gerking. 2024. Kfeature: Rendering the Kconfig System into Feature Models. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS).* ACM, 134–138. doi:10.1145/3634713.3634731