Can SAT Solvers Keep Up With the Linux Kernel's Feature Model?

Elias Kuiter
kuiter@ovgu.de
University of Magdeburg
Magdeburg, Germany

Urs-Benedict Braun
urs-benedict.braun@st.ovgu.de
University of Magdeburg
Magdeburg, Germany

Thomas Thüm
thomas.thuem@tu-braunschweig.de
TU Braunschweig
Braunschweig, Germany

Sebastian Krieter
sebastian.krieter@tu-braunschweig.de
TU Braunschweig
Braunschweig, Germany

Gunter Saake
saake@ovgu.de
University of Magdeburg
Magdeburg, Germany

Abstract

The Linux kernel is a highly relevant, yet also highly configurable software system. Kernel developers keep track of this configurability in a feature model, which defines the features of Linux and their dependencies. To support kernel developers in various activities, many (semi-)automated product-line analyses have been proposed over the years. Under the hood, these analyses can often be computed with SAT solvers. Yet, the Linux kernel has constantly been growing in complexity for decades, which increasingly hampers its efficient analysis. At the same time, SAT solvers have been improving in performance for decades, which eases analysis. In this paper, we investigate empirically whether SAT solvers can keep up with the Linux kernel's feature model. To this end, we analyze historic feature models of Linux with historic SAT solvers from several sources. We find that SAT solvers are generally not able to keep up with Linux. Even the optimal SAT solver is slowing down by 10% every year, meaning that its performance halves every seven years. We conclude that the Linux kernel will become increasingly difficult to analyze if its growth is not counteracted.

CCS Concepts

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

Keywords

feature modeling, satisfiability solving, Linux kernel

ACM Reference Format:
Elias Kuiter, Urs-Benedict Braun, Thomas Thüm, Sebastian Krieter, and Gunter Saake. 2025. Can SAT Solvers Keep Up With the Linux Kernel's Feature Model?. In *Proceedings of 48th International Conference on Software Engineering*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

48th International Conference on Software Engineering, Rio de Janeiro, Brazil

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Configurable software systems play a crucial role in today's software landscape. They are used in a wide range of application areas, such as operating systems, embedded systems, and enterprise software [111]. The *Linux kernel* [122], for example, is a widely used and highly configurable software system [109], which is being developed in form of a feature-oriented *software product line* [9]. Today, Linux has more than 20,000 features (cf. Section 2), which can be flexibly combined to meet the specific needs of different hardware and use cases [81]. This flexibility likely contributed to the success of Linux, which is the dominant operating system on servers,[1] supercomputers,[2] and smartphones,[3] and also increasingly being used in embedded devices (e.g., for the internet of things).[4]

While it increases its applicability, the high configurability of the Linux kernel also makes it highly complex to maintain, improve, and validate. For example, a bug may occur in some, but not all configurations of the kernel, which hampers bug detection [1, 25, 132]. To combat and reason about the complexity arising from the kernel's configurability, a variety of product-line analyses have been proposed for and applied specifically to Linux [1, 6, 45, 93, 96, 99, 116, 132]. Many of these analyses rely on *satisfiability* (SAT) *solving* underneath, which is well-established for reasoning about features in product lines and their dependencies [12, 13, 95, 120]. It has been found that, despite being NP-hard, SAT solving usually performs reasonably well when analyzing product lines [63, 86, 92].

However, as most software systems, the Linux kernel is not static. Indeed, it has been continuously evolving for over three decades, all the while growing continuously and considerably in complexity [81, 89] (cf. Figure 1a), with future growth to be expected according to market analyses.[5] This raises the question: *Can SAT solvers keep up with the increasing complexity of the Linux kernel?*

In this paper, we aim to answer this question empirically by analyzing historic versions of the Linux kernel with matching historic versions of SAT solvers. Thus, we aim to determine whether there is a need to further improve SAT solving algorithms, or even to proactively reduce the complexity of the Linux kernel, so that product-line analyses continue to scale in the future.

Overall, we contribute the following in this paper:

[1] https://w3techs.com/technologies/overview/operating_system (April 2025)
[2] https://www.statista.com/statistics/565080/ (June 2023)
[3] https://www.statista.com/forecasts/997100/ (9,241 participants, December 2024)
[4] https://www.businessresearchinsights.com/market-reports/--108687 (2024)
[5] Further market reports: 115306, 108687, 103037

- We collect yearly snapshots of the Linux kernel from the time frame 2002–2024, extracting all features and their dependencies.
- We then collect historic SAT solvers from the same time frame, which represent the leading research in SAT solving at their time.
- Moreover, we collect all SAT solver versions integrated into a well-known product-line analysis tool, which reflects the typical usage of SAT solving in industrial practice.
- We then evaluate the runtime of each collected SAT solver on each extracted version of Linux, discussing the results. We publish our automated evaluation in a reproduction package.[6]

Among others, we derive insights regarding the following questions: Which SAT solver can be recommended for analyzing configurability in the Linux kernel? Does the chosen SAT solver matter in practice? Are SAT solvers sensitive to the kernel's evolution? And will we be able to analyze the Linux kernel in 10–20 years?

The answers to these questions have potential to impact software engineering researchers and practitioners working with the Linux kernel and comparable product lines. In addition, our evaluation sheds new light on the evolution of SAT solving performance in practice, which is relevant for automated reasoning researchers.

2 Background

To begin with, we recap and give evidence for how both the Linux kernel and SAT solvers have developed over the past decades. We also discuss the relevance of SAT solving for product-line analyses, and how these analyses are applied to the Linux kernel, specifically.

2.1 Growth of the Linux Kernel

It is widely understood that evolving software systems have the tendency to grow. To capture this notion, Lehman [83, 84] postulated eight well-known laws of software evolution. These laws, while originally based on anecdotal evidence, have been empirically confirmed to varying degrees [52, 53, 61, 81, 110, 134]. One almost universally confirmed law is that of *continuing growth* of real-world software systems, which is typically measured by size metrics of core assets (e.g., source lines of code [103]). This trend for growth can also be observed in software product lines [33].

The Linux kernel [109, 122], in particular, is a prime example of a product line that has continually been growing over time since 1991. In particular, Lehman's laws of continuing change and growth have been empirically confirmed on the Linux kernel [52, 61, 81, 89]. We illustrate this growth in Figure 1a, where we show how two key complexity metrics of the Linux kernel have evolved over the course of two decades: its source lines of code[7] and its number of features.[8] *Source lines of code* are a common metric for software size in general [103]. The number of *features* (or *configuration options*) is a metric for the configurability [13, 116] of a product line. It is measured on a *feature model* [9, 69], which systematically specifies the product line's features and their dependencies. Indeed, both metrics have grown approximately linearly over time (Pearson's $r \geq 0.98, p < 0.001$) [81].[9] This indicates that the Linux kernel has steadily been growing more complex, gaining about one million source lines of code and over 800 new features per year since 2002. Consequently, the kernel's feature model has also grown more complex, as new features also typically come with new, potentially complex dependencies [73, 74], which must be documented.

The Linux kernel's feature model has grown considerably in complexity over the last two decades (cf. Figure 1a).

2.2 Product-Line Analyses

To cope with the complexity of product lines such as the Linux kernel, a wide variety of *product-line analyses* have been proposed [13, 26, 114, 120], and new ones keep emerging [95]. These analyses support various activities across all development phases of software development, such as parsing [71], dead-code analysis [116], type checking [87], testing [125], refactoring [39], feature-model evolution [97, 121] and slicing [2], and economical estimations [28]. Thus, product-line analyses play a major role in supporting the development and quality assurance of product lines.

Product-line analyses can often be reduced to SAT problems that also operate on the feature model. The feature model on its own already gives rise to a surprising variety of relevant *feature-model analyses* [13, 49, 114], such as consistency checking [13], decision propagation [9], or t-wise sampling [125]. Most feature-model and product-line analyses rely on encoding the feature model as a *propositional formula* [12, 13]. To this end, features are mapped to Boolean variables, and feature dependencies are encoded in *conjunctive normal form* (CNF) [12, 80]. The CNF representation of a feature model is then fed into off-the-shelf SAT solvers [12, 92]. Building on this idea, more complex analyses may add additional conditions to the SAT query (e.g., concerning the code regions under analysis) [9, 13, 120] or solve many slightly different SAT queries [113, 133]. Well-known modern product-line tools (e.g., FeatureIDE [91] or Flama [50]) heavily rely on these SAT-based reasoning techniques to enable powerful analyses.

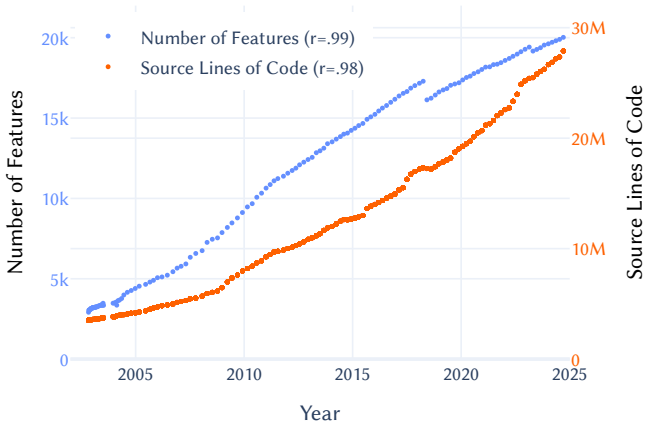
Analyses of product lines are crucial for their successful development and quality assurance, and often reduced to SAT solving.

2.3 Advances in SAT Solving

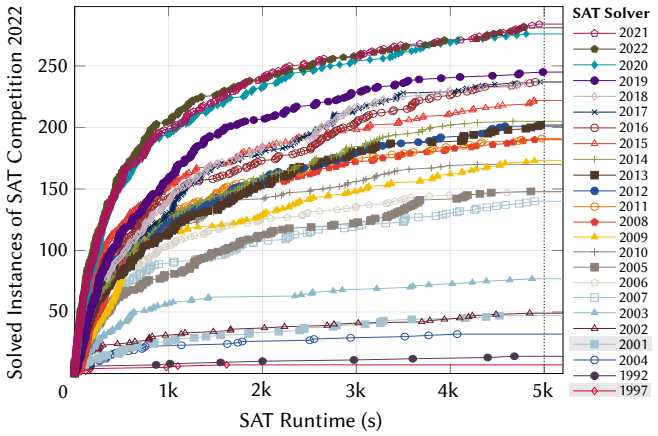
SAT solvers, which are instrumental to product-line analysis, have been continually optimized over the last two decades [19, 42, 43, 66]. In theory, SAT is an NP-hard problem with exponential worst-case time complexity if P ≠ NP [67]. However, several algorithmic breakthroughs have made SAT solving tractable [19, 43] on many theoretical and practical problems [27]: That is, in the 1960s, the basic DPLL backtracking procedure [30, 31] was introduced. In 1997, the solver GRASP [108] extended DPLL towards conflict-driven clause learning (CDCL), which still is the foundation of most modern SAT solvers. In 2001, the solver Chaff [94] then introduced the efficient branching heuristic VSIDS, variants of which are still in use today. Besides these breakthroughs, small and steady improvements have been made in SAT solving over the years [19, 66].

The algorithmic improvements in SAT are driven and documented by yearly SAT competitions[10a] [11, 47] conducted by the

[6] https://doi.org/10.5281/zenodo.16084321
[7] Counted with: `git clone https://github.com/torvalds/linux && cloc linux`
[8] Counted according to the procedure described by Kuiter et al. [81].
[9] In 2018 and 2023, obsolete architectures were removed. This temporarily slowed down kernel growth (see `https://lwn.net/Articles/{751885, 920259, 950466}`).



(a) Evolution of the Linux Kernel's Complexity [81]



(b) Evolution of SAT Solving Performance (adapting Biere et al. [19])

Figure 1: While the Linux kernel has grown more complex over the last few decades, SAT solvers have grown more efficient.

SAT association.[10b] In these competitions, solvers [20] compete against each other in different tracks on blinded benchmarks, and the *single best solver* [130] of each year is awarded. In Figure 1b, Biere et al. [19] show how the performance of the single best solver has evolved over two decades.[11] In this cactus plot, higher generally means better. That is, solvers that are high on the y-axis are able to solve many instances of the SAT competition benchmark 2022 in a given time frame on the x-axis. Indeed, more recent solvers are overall much more time-efficient on this benchmark than older ones, with a new leap in performance every 3–5 years [19]. This is evidenced by the legend of Figure 1b, which is sorted by the number of instances solved in 5,000 seconds (i.e., solver performance). At the same time, the legend also lists solvers mostly in historical order, and with large performance gaps in between. Notably, these leaps are purely due to algorithmic advances on the software side, as Biere et al. conducted this experiment on fixed hardware [19].

SAT solvers have become considerably more time-efficient over the last two decades (cf. Figure 1b).

2.4 SAT-Based Analysis of the Linux Kernel

Software engineering researchers are leveraging the potential of efficient SAT solving for analyzing configurability in the Linux kernel. That is, for almost 15 years, product-line analyses have been applied to Linux [1, 6, 45, 93, 96, 99, 116, 132]. Analyzing Linux remains a challenging task—not only due to the kernel's complexity (cf. Figure 1a), but also because of its feature model's structure.

First, the Linux kernel is currently being developed for 20 different CPU architectures at the same time [23, 81], which are modeled and configured independently to some degree. As some of these architectures differ considerably in terms of their feature model [34, 35, 81, 104], the correct architecture must be selected carefully when preparing an analysis.

Second, the feature model of Linux is specified in KConfig, a domain-specific configuration language that was developed for the kernel in 2002 [7, 77, 118]. To compute any product-line analysis, we first have to extract a suitable propositional formula (cf. Section 2.2) from all of its KConfig specifications [12]. However, such an extraction is far from trivial, as the KConfig language has complex semantics, undocumented edge cases, and non-propositional modeling constructs [40, 45, 99, 107]. Nonetheless, several KConfig extraction approaches have been proposed [15, 40, 45, 70, 98, 116, 126], the most influential being KConfigReader [70] and KClause [98]. Both of these extractors have repeatedly been applied to analyze the propositional fragment of the Linux kernel, which accounts for ≈ 96% of its features [15, 81, 100].

Third, the extracted feature-model formulas are generally not in CNF, which is the required input format of most SAT solvers [60, 62, 65, 66]. Thus, the kernel's feature-model formulas must be transformed into CNF first. While this step is often disregarded, it is nontrivial for feature models such as the Linux kernel's, which often contain complex non-CNF constraints [74]. Consequently, this step can considerably impact the efficiency of SAT solving [80].

To analyze the Linux kernel's feature model, one must choose a suitable architecture, extractor, and CNF transformation.

3 Problem Statement

In this paper, we aim to improve our understanding of how the practical computational complexity of SAT-based analyses evolves in the long term. We choose to investigate the Linux kernel's feature model as a case study. We do this for several reasons: First, the Linux kernel is a highly relevant, infrastructure-critical software system, whose flexibility is integral to its permeating usage throughout computing today (cf. Section 1). Thus, the scalability of analyses on Linux can have direct consequences for researchers and kernel developers who rely on such analyses. Second, the Linux kernel is often investigated and hypothesized about in the literature [81], and therefore also an interesting case study for researchers. Third, the Linux kernel is likely the only product line that is suited for

[10](a) https://satcompetition.github.io/, (b) https://satassociation.org/
[11]An interactive version of Figure 1b is also available online: https://cca.informatik.uni-freiburg.de/satmuseum/

a longitudinal study. This is because its feature model has a long, consistent history, dating back as far as 2002. Incidentally, this perfectly matches the documented history of SAT solving. We are not aware of any other published product line that fits this criterion.

So, given the kernel's relevance, how difficult are SAT-based analyses on Linux today? It is already known that, on many feature models, SAT-based analysis is comparably "easy" [63, 86, 92]. Even on the complex Linux kernel, "a typical satisfiability check requires half a second on a standard computer" (as of 2010) [87]. However, this "easiness" only refers to a single SAT query on a feature model [79], which performs a simple consistency check [9, 13]. In fact, many analyses require repeated SAT queries to analyze feature models [113, 133]. For these analyses, a median SAT query runtime of 10 versus 100 milliseconds might make the difference in scalability: For instance, all variants of the BusyBox system [127] can be parsed and type checked with SAT assistance in roughly one hour [71]. However, merely parsing the much larger Linux kernel with SAT assistance already takes 85 hours [71], and type checking all of its variants has not been successful yet. Moreover, several analyses [102, 114] rely on specialized solvers (e.g., #SAT [112] or AllSAT [48]) or algebraic reasoning (e.g., slicing [2] or differencing [4]). While these build on the same principles as SAT solving (e.g., exhaustive DPLL [30, 31]), they are even harder than SAT [124]. Today, specialized solvers and algebraic reasoning do not scale to Linux and other large product lines [80, 99, 102, 112, 113, 119]. In summary, many nontrivial analyses are challenging or even intractable on Linux. Compared to these analyses, the runtime of a single SAT query is a pragmatic and foundational metric for the computational complexity of a feature model [79].

Crucially, the runtime needed for a SAT query on a feature model may change as the feature model evolves over time. In fact, the Linux kernel has evolved towards having the most configurable feature model currently known, and it is still growing (cf. Figure 1a) [81]. This growth affects the efficiency of many analyses, and it may even affect their general tractability. For example, counting the number of valid configurations of Linux with a #SAT solver is currently limited to decades-old kernel versions [68, 81]. This begs several questions: Is this situation going to improve in the future? That is, can software engineering researchers continue to simply rely on the steady progress made by the automated reasoning community (cf. Figure 1b)? If not, do we need to improve analysis efficiency with new algorithmic breakthroughs in product-line analyses and SAT solving? Or should we try to counter the growth of the kernel's feature model by advocating for variability reduction [5, 8, 117] (e.g., by removing features from the kernel)?

Our goal is to journey through the past two decades to investigate these questions, and to address the following problem statement:

> Is the Linux kernel's feature model growing more complex at a faster pace than SAT solvers become more efficient? In other words: *Can SAT solvers keep up with the Linux kernel's feature model?*

With our study, we aim to gather empirical evidence for whether SAT-based analysis of evolving feature models is becoming easier or more difficult over time. We investigate the Linux kernel as one particularly relevant and well-suited example of long-term feature-model evolution. We measure runtime of a simple SAT query, which, besides being a rough indicator of computational complexity, is also a necessary prerequisite for most nontrivial analyses on feature models. With such a simple query, we do not seek to push SAT solvers to their limits or explore the boundaries of their scalability. Indeed, related questions have already been studied for #SAT and knowledge compilation [81, 102, 112, 119]. Instead, we complement these previous studies on upper bounds (i.e., "Do complex analyses scale on Linux?") with new insights on *lower* bounds (i.e., "Can we even expect the simplest analysis to keep scaling?").

By studying this problem, we aim to help software engineers to anticipate or even mitigate future challenges in product-line analyses. Moreover, our results could provide valuable insights for the automated reasoning community, which has, to the best of our knowledge, not studied evolving propositional formulas yet.

4 Towards Evaluating SAT Solvers on Linux

To answer the problem statement, we aim to empirically evaluate the evolution of SAT solving performance on the Linux kernel's feature model. This requires careful planning, so we can ensure the validity of our conclusions. We proceed to describe and justify two complementary experiments and how we intend to conduct them.

4.1 Defining the Experiments

We envision two experiments ①–② to study the problem statement from both an academic and an industrial point of view.

4.1.1 Academic Point of View. In experiment ①, we aim to investigate how the leading research in SAT solving has evolved over the last two decades. That is, we want to evaluate the influence of new or improved SAT solving algorithms, data structures, and optimizations, all of which represent key knowledge of the SAT community at a given time. The best-performing advancements are at some point reflected in the implementations of SAT solvers submitted to the SAT competition. In principle, each submitted solver may contain interesting innovations. However, evaluating all 15–50 solvers submitted per competition would take years of evaluation time. Instead, we focus on evaluating the winning solvers from this competition in experiment ①. By winning, these solvers evidently contributed key innovations in SAT solving over the years, making them a reasonable choice for researchers and practitioners alike.

4.1.2 Industrial Point of View. In experiment ②, we aim to focus on the evolution of SAT solving performance in practice. This point of view is distinct from experiment ①, as the solvers in widespread analysis tools are rarely updated to reflect the latest research progress. This is because integrating new solvers is often infeasible due to the use of non-standardized APIs.[12] Even just upgrading a solver to a new version requires effort (e.g., to test for potential regressions). Here, we investigate FeatureIDE [91], which was released in 2009, and which has evolved into a well-known [14] and industrial-grade [46, 56] product-line tool ever since. For analyses, FeatureIDE relies on the Java-based SAT solver `Sat4j` [82]. While `Sat4j` has not been particularly successful in the SAT competition, it powers dependency solving in Eclipse and many other tools [82]. For conducting this experiment, FeatureIDE (powered by `Sat4j`)

[12] While standardized interfaces like IPASIR have been proposed [11] and evolve [38], they do not cover all solver features used in tools like FeatureIDE, such as MUS [88].

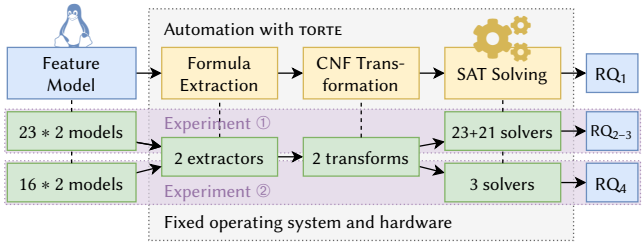


Figure 2: Evaluation pipeline for our experiments.

Table 1: Variables considered in our experiments.

Variable	Type	Values	RQ
Feature Model			
Version	Independent	*(cf. Table 2a)*	RQ_{2-4}
Architecture	Independent	`x86`, `arm`	$RQ_{1.1}$
Formula Extraction	Independent	KConfigReader, KClause	$RQ_{1.2,\ 2-4}$
CNF Transformation	Independent	KConfigReader, Z3	$RQ_{1.3,\ 2-4}$
SAT Solving			
Solver	Independent	*(cf. Table 2b)*	RQ_{2-4}
Compiler (only for ①)	Confounding?	Competition: `gcc/g++ ??`	$RQ_{1.4}$
	Control	Museum: `gcc/g++ 9.4.0`	
Analysis	Control	SAT query (20min timeout)	–
Deep Variability			
Operating System	Control	Ubuntu `22.04` (in Docker)	–
Hardware	Control	Intel Xeon E5-2630 2.4 GHz	–
Iteration	Independent	5 * 5 * 3 iterations	$RQ_{1.5,\ 2}$
SAT Runtime	Dependent	*(cf. Section 5)*	RQ_{1-4}

We *vary* independent variables, *fix* control variables, and *measure* dependent variables.

is the only sensible candidate, as we are not aware of any other product-line tool with such a long history and industry relevance.

4.2 Designing an Evaluation Pipeline

Moving on, we describe how we intend to conduct the experiments ①–② by designing a pipeline of successive steps. In both experiments, we "travel back in time" by analyzing historic feature models of the Linux kernel using SAT solvers available *at that time*. To ensure a fair comparison, we have to carefully consider several experiment variables, which we list in Table 1. We vary some of these variables (to allow for time travel), while fixing others (to only measure how SAT solving evolved, and nothing else). Together, these variables form our evaluation pipeline as shown in Figure 2.

4.2.1 Feature Model. First, we select the feature models to consider in both experiments (see in Figure 2 and Table 1–2a).

Version In experiment ①, we evaluate the entire development history of the mainline Linux kernel [29] ever since 2002, when its feature model was first specified. As shown in Table 2a, we evaluate one feature model per year, as we only have one SAT solver for each yearly-held SAT competition to pair it to. In each year, we choose the most recent version of Linux (released in the last quarter), which we can then pair with the single best SAT solver from that year's SAT competition (which is usually held in the summer).

In experiment ②, we can only investigate SAT solvers from 2009 onwards (as explained below), which is why we do not consider feature-model versions from 2002 to 2008.

Table 2: Evaluated feature models and SAT solvers.

Year	Version	Year	Version
'02	`2.5.53`	'14	`3.18`
'03	`2.6.0`	'15	`4.3`
'04	`2.6.10`	'16	`4.9`
'05	`2.6.14`	'17	`4.14`
'06	`2.6.19`	'18	`4.20`
'07	`2.6.23`	'19	`5.4`
'08	`2.6.28`	'20	`5.10`
'09	`2.6.32`	'21	`5.15`
'10	`2.6.36`	'22	`6.1`
'11	`3.1`	'23	`6.6`
'12	`3.7`	'24	`6.12`
'13	`3.12`	*'25*	*N/A*

In experiment ①, we analyze all years. In experiment ②, we only analyze years '09–'24. In each analyzed year, we consider two architectures (`x86`, `arm`). We only consider the mainline kernel of Linux.

(a) Feature Models

	Year	Solver
Experiment ①	'02, '04	`zChaff`†, `Limmat`‡
	'03	`Forklift`,† `BerkMin`‡
	'05	`SatELiteGTI`
	'06, '08	`MiniSat`
	'07	`RSat`
	'09	`PrecoSAT`
	'10	`CryptoMiniSat`
	'11–12	`Glucose`
	'13–14	`Lingeling`
	'15	`abcdSAT`
	'16–19	`MapleSAT`
	'20–22, '24	`Kissat` ('24†)
	'23	`SBVA-CaDiCaL`†
Exp. ②	'09–11	`Sat4j (2.1.0)`
	'11–14	`Sat4j (2.3.1)`
	≥ 2014	`Sat4j (2.3.5)`

†SAT Competition only ‡SAT Museum only

(b) SAT Solvers

Architecture Moreover, we have to consider which architectures to evaluate (cf. Section 2.4) [81]. Out of the 20 architectures available today, only nine have been in development since 2002. Here, we opt to evaluate the architectures `x86` (formerly `i386`) and `arm`. Together, they cover most consumer devices running Linux today [54].

4.2.2 Formula Extraction. In order to compute SAT-based analyses on the Linux kernel, we first have to extract propositional formulas from its KConfig specifications (cf. Section 2.4). Here, we rely on the well-known tools KConfigReader [70] and KClause [98] for that extraction. Each tool makes different trade-offs regarding the implemented semantics (e.g., they encode `tristate` features differently) [37, 45, 81, 98], which is why we choose to evaluate and compare both of them in our experiments.

4.2.3 CNF Transformation. The extracted feature-model formulas must then be transformed into CNF (cf. Section 2.4). Here, we choose to evaluate two influential CNF transformation algorithms proposed by Plaisted-Greenbaum [101] (implemented in KConfigReader [70]) and Tseitin [123] (implemented in Z3 [32]). We choose these implementations because they are often used in the literature [71, 97, 99]. We dismiss transformation by logical equivalences, which is the textbook algorithm for CNF transformation [24]. We do this because equivalence transformation does not scale to the complexity of the kernel's feature model [80].

4.2.4 SAT Solving. We select, compile, and run SAT solvers in our experiments as follows (see in Figure 2 and Table 1–2b).

Solver For experiment ①, we collect 23 historically significant SAT solvers written in C/C++. As we aim to investigate the evolution of the leading research in SAT solving throughout time, we choose to evaluate solvers that have been awarded in the SAT competition[10] [11, 47]. This competition has been conducted yearly ever since 2002. Surprisingly, this happens to match the inception of the Linux kernel's feature model, so we can investigate the competition's entire history. We choose the single best solver in each competition's main track (i.e., gold medal winners in sequential SAT

+ UNSAT, ranked by their PAR-2 score [47]). We list all winning solvers in Table 2b and refer to them as *SAT competition solvers*. For brevity, we group solvers with similar names in families, but each solver makes unique contributions. While these winning solvers are not guaranteed to include "the single best" reference solver for Linux, identifying such a solver is not our goal here. Instead, we choose winning solvers to represent a wide range of DPLL-based SAT solving techniques [19], which allow us to explore the evolution (not optimization) of SAT performance.

In experiment ②, we want to investigate the evolution of the `Sat4j` [82] solver integrated into the product-line tool FeatureIDE [91]. As FeatureIDE has only upgraded this solver twice since its public release in 2009, we only consider its three used versions of `Sat4j` here (cf. Table 2b).

Compiler Our collection of SAT competition solvers for experiment ① may have a weakness, as we use the original solver binaries that have been submitted to the SAT competition. Consequently, these binaries were compiled with C/C++ compilers of varying age, which makes the compiler a potentially confounding variable [19]. Fortunately, Biere et al. [19] have published a dataset of 21 compiler-controlled SAT solvers, known as the SAT museum.[11] In this dataset, they have obtained the original source code for all winning solvers and recompiled them with a single fixed compiler. While these *SAT museum solvers* (cf. Figure 1b) mitigate the compiler as a confounding variable, they require extensive patches to the solvers' source code and only cover the SAT competition until 2022. We decide to evaluate both solver datasets. This way, we can assess whether the compiler is indeed a confounding variable, and we can combine the advantages of both datasets.[13]

In experiment ②, we do not control the Java compiler for `Sat4j` in the same way. While this may influence our results, this experiment is deliberately conducted in a less controlled environment to study the SAT solver in situ (i.e., in its natural setting).

Analysis In both experiments, we perform a single SAT query with a 20-minute timeout on each feature model by calling the solver on the transformed CNF. This query performs a simple consistency check (or void analysis) that can reveal grave modeling errors [9, 13]. We focus on the simple SAT query here for several reasons: First, it is a necessary prerequisite analysis on feature models, so it already forms the basis for more complex analyses (cf. Section 2.2) [79, 113, 133]. Thus, it also constitutes a lower bound for the scalability of SAT solving on Linux, indicating whether *any* feature-model analyses can keep up with the growing complexity, rather than representing the scalability of a *specific* (and arbitrarily selected) complex feature-model analysis. Second, the single SAT query is the only interface shared by all investigated solvers, as more powerful APIs have only been proposed in recent years. For example, the standardized incremental solving interface IPASIR [11] (required by many algorithms) has only been proposed in 2015, which disqualifies many older solvers. Even further, several advanced analysis algorithms rely on completely unstandardized solver APIs (e.g., MUS extraction [88]). Third, more advanced solvers (e.g., #SAT [41]) or analyses (e.g., core and dead features [9, 21, 64] or atomic sets [57, 106]) often only date back a few years, which hampers a rigorous evolutionary analysis. Finally, advanced analyses would also be too complex to evaluate on a large scale, with some even being infeasible on Linux (e.g., #SAT [81]).

4.2.5 Deep Variability. Deep variability refers to all external layers that may affect the reproducibility of research evaluations [3, 85]. Some of these layers may affect our experiments, which we discuss.

Operating System The operating system (along with its shared libraries) may influence the results of research evaluations [51]. In both experiments, we run all solvers in a Ubuntu environment in the same `amd64` Docker container. This is necessary for reproducibility, as some SAT solvers rely on specific versions of shared libraries. As the host system, we use AlmaLinux 8.10, based on Linux 4.18.0.

Hardware Similarly, hardware may influence absolute measurements [42]. Here, we run all solvers on a single high-end server, which ensures a fair comparison (an Intel Xeon E5-2630 CPU with 8 cores, 2.4 GHz, and 1 TiB RAM). In addition to this high-end machine, we performed smaller pre-experiments on a consumer laptop, which lead us to the same conclusions (an AMD Ryzen 7 6800U CPU with 8 cores, 2.7 GHz, and 16 GiB RAM).

Iteration To account for nondeterminism and fluctuating measurements, we perform several iterations of each step in our evaluation pipeline. That is, we run five iterations of each extraction and transformation step, as they are nondeterministic and may affect the subsequent SAT solving step (e.g., due to differences in clause order [22] and auxiliary variables [80]). We also run three iterations of each SAT query, as we expect SAT runtime to vary slightly.[14]

4.2.6 SAT Runtime. As the target metric for both experiments, we measure each SAT solver's total runtime (including parsing and preprocessing) on each feature-model formula. We are not interested in the runtime or accuracy of the extraction and transformation steps, which are already studied elsewhere [80, 81].

5 Empirical Evaluation

In the following, we conduct the evaluation envisioned in Section 4. We address its research questions, execution, results, and insights.

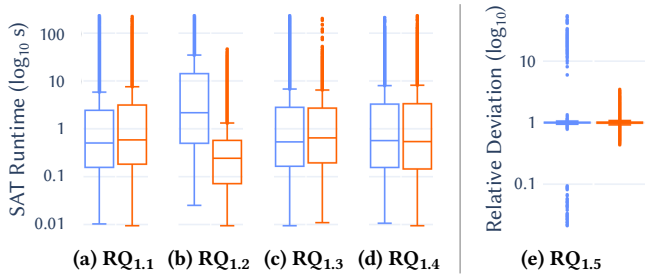
5.1 Research Questions

We pose four research questions, which cover various aspects of the experiments ①–② as shown in Figure 2 and Table 1.

- **RQ$_1$** How is SAT runtime influenced by the …
  - **RQ$_{1.1}$** … architecture of the Linux kernel?
  - **RQ$_{1.2}$** … extractor for feature-model formulas?
  - **RQ$_{1.3}$** … CNF transformation of feature-model formulas?
  - **RQ$_{1.4}$** … C/C++ compiler used to build SAT solvers?
  - **RQ$_{1.5}$** … iterated execution of any experiment step?
- **RQ$_2$** How does the runtime of historic SAT solvers evolve over the history of the Linux kernel's feature model?
- **RQ$_3$** How does SAT runtime evolve when, in year y, we use …
  - **RQ$_{3.1}$** … the most recent winning solver ($= y$)? *(Current)*
  - **RQ$_{3.2}$** … the fastest solver at that time ($\leq y$)? *(Optimal)*
  - **RQ$_{3.3}$** … the fastest solver as of today (≤ 2024)? *(Historic)*
- **RQ$_4$** How does SAT runtime of a well-known and widely used product-line analysis tool evolve? *(Pragmatic)*

[13] In Table 2b, the SAT solvers in 2002 and 2003 differ between both datasets, which Biere et al. [19] attribute to challenges in obtaining and patching these old solvers.

[14] Exemplified here: https://www.msoos.org/2017/02/non-reproducible-maplecomsps/



For the **architecture**, **formula extraction**, **CNF transformation** ($RQ_{1.1–1.3}$), and **compiler** ($RQ_{1.4}$), we compare the two values listed in Table 3, respectively. For the **iteration** ($RQ_{1.5}$), we isolate 2024's `Kissat` (left) from all the other SAT solvers (right).

Figure 3: How do the variables influence our results (RQ_1)?

In RQ_1, we identify the relevant variables (cf. Table 1) for subsequent research questions. In RQ_2–RQ_3, we consider historic SAT solvers, addressing experiment ①. Finally, in RQ_4, we consider the evolution of the SAT solver in FeatureIDE for experiment ②.

5.2 Experiment Execution

We integrate our evaluation into our open-source tool TORTE [78], which is a Docker-based declarative workbench for automated and reproducible feature-model experiments. The tool bundles and adapts all tools required for our evaluation (cf. Table 1). In particular, we modify KConfigReader and KClause such that historic versions of Linux can be extracted consistently and reliably. For each selected solver (cf. Table 2b), we integrate binaries from the SAT competition [11, 47] and SAT heritage [10] initiatives. We also integrate the SAT museum solvers that Biere et al. [19] patched and recompiled. We publish the tool and all our data (including interactive visualizations) in a reproduction package.[6]

We execute our experiment with TORTE, which requires roughly six weeks (more than five weeks spent on roughly 500,000 SAT queries in total). We are able to successfully extract formulas for all feature models. With a single exception,[15] we also succeed at transforming the extracted formulas into CNF. For SAT solving, the 20-minute timeout is never hit. Moreover, all feature models are found to be satisfiable by every SAT solver, as expected for the Linux kernel, which should not include major contradictions [116].

5.3 Results and Discussion

We proceed to describe and discuss the results of our evaluation. We start by assessing the influence of evaluated variables (cf. Table 1).

5.3.1 $RQ_{1.1}$: Architecture. In Figure 3a–3d, we show box plots for the logarithmic runtime of all SAT solvers on all feature models. We visualize $RQ_{1.1}$ in Figure 3a, where we group all measurements by the analyzed architecture of the Linux kernel.

Results There is a significant, but negligible difference between the two considered architectures, `x86` and `arm`, with regard to their required median SAT runtime (cf. Table 3, which shows test results).

Discussion It is known that `x86` and `arm` cover different (i.e., architecture-specific) features of the Linux kernel [34, 35, 81, 104]. Yet, the SAT runtime of both architectures is very similar, which indicates that both architectures' feature models are of similar computational complexity. Due to the negligible differences, we only discuss results for the `x86` architecture in $RQ_{2–4}$.[16]

5.3.2 $RQ_{1.2}$: Formula Extraction. In Figure 3b, we visualize $RQ_{1.2}$, grouping all measurements by their KConfig formula extraction.

Results The SAT runtime on feature-model formulas extracted with KConfigReader and KClause differs significantly (large effect size). With KConfigReader, SAT solving is noticeably slower.

Discussion The difference in SAT runtime is due to the different semantics encoded by both extractors [37, 45, 81, 98]. Most noticeably, KClause treats commonly occurring `tristate` features as `bool`, which makes its feature models much easier to solve. As the desired semantics depend on the concrete analysis use case, no extractor is clearly preferable. Thus, we discuss both in $RQ_{2–4}$.

5.3.3 $RQ_{1.3}$: CNF Transformation. Next, we visualize $RQ_{1.3}$ in Figure 3c, grouping all measurements by their CNF transformation.

Results Median SAT runtime differs significantly depending on the CNF transformation (cf. Table 3). While CNFs transformed with KConfigReader are, in the median, solved slightly faster than those transformed with Z3, they also have a 72.6% larger variance in runtime. In fact, we observe two opposing effects: Formulas extracted with KConfigReader are solved faster if transformed with Z3 (effect size $r = .06$), while formulas extracted with KClause are solved faster if transformed with KConfigReader ($-.11$).

Discussion When distinguishing extractors, we find two opposing effects regarding CNF transformation. These effects appear to nearly cancel each other, which explains the negligible difference in the median. Thus, the preferable CNF transformation depends on the chosen extractor. As we aim to discuss both extractors in $RQ_{2–4}$, we must consequently also discuss both CNF transformations.

5.3.4 $RQ_{1.4}$: Compiler. In Figure 3d, we visualize $RQ_{1.4}$, grouping all measurements by the source of their SAT solvers' binaries. The SAT competition uses compilers of varying age, while the SAT museum pins all solvers to a single compiler version (cf. Table 1). We exclude several years from the comparison (i.e., 2002–2003, where solvers differ; and 2023–2024, which are not yet in the museum).

Results Regarding median SAT runtime, there is a significant, but negligible difference between the SAT competition and museum solvers. However, when we distinguish years, five museum solvers are significantly faster than their competition counterparts: in 2004 (effect size $r = .12$), 2005 (.02), 2011, 2012 (.06), and 2021 (.02). At the same time, 2006's museum solver is slower than its counterpart ($-.04$). For all remaining solvers, there is no significant difference.

Discussion On the one hand, some SAT museum solvers (compiled recently) are indeed faster than their SAT competition counterparts (compiled years ago). On the other hand, many solvers are not, and effect sizes are small to negligible. Moreover, the median difference is negligible. Thus, we conclude that the compiler is not generally a confounding variable when evaluating SAT solvers on the Linux kernel. In $RQ_{2–4}$, we combine the best of both solver datasets: That is, we use the SAT museum solvers for 2002–2022 (as they control the compiler), and the SAT competition solvers for

[15] For Linux `v6.12` in the `arm` architecture, the formula extracted by KClause cannot be transformed into CNF by KConfigReader due to a reproducible Scala exception.

[16] Wherever we omit results for brevity, they are available in our reproduction package.[6]

Table 3: Statistical comparison of experiment variables (RQ_1).

Variable	Compared Values	p-Value	Eff. Size	Rel. Var.
Architecture	`x86`, `arm`	.000	.04	11.4%
Formula Extraction	KConfigReader, KClause	.000	−.56	99.2%
CNF Transformation	KConfigReader, Z3	.000	−.02	72.6%
Compiler	Competition, Museum	.000	.01	2.5%

For testing significance, we apply a Mann-Whitney U test with $\alpha = .005$, reporting $r = z/\sqrt{N}$ as effect size, as well as relative variance (i.e., how the unbiased variance s^2 compares for each variable). All results are statistically significant (i.e., $p < .005$).

2023 and 2024 (so we can evaluate the most recent SAT solvers). We do not show explicit visualizations of both solver datasets here, as they are very similar and yield no new insights.[16]

5.3.5 $RQ_{1.5}$: Iteration. Finally, in Figure 3e, we visualize $RQ_{1.5}$ with the logarithmic relative deviation from the median SAT runtime.

Results Half of all our measurements are within 2.2% of the median SAT runtime. The `Kissat` solver from 2024 has worst-case outliers of up to 55 times the median SAT runtime. All other solvers have worst-case outliers of up to 3.5 times that runtime. For a single extraction and transformation step, measurements are within 0.7%, and the above factors drop to 1.2 and 1.8.

Discussion For most solvers, the differences in SAT runtime are negligible. Noticeable differences appear only due to the extraction and transformation steps, which are nondeterministic. `Kissat` from 2024, in particular, is sensitive to said steps, which is why we isolate it in Figure 3e. We manually inspected how these differences affect our following results and discussions (i.e., by plotting `min` and `max` values). We came to identical conclusions, which is why we stick to discussing and visualizing the median SAT runtime in the following.

5.3.6 RQ_2: Evolution of SAT Runtime. In RQ_2, we investigate how SAT runtime evolves in the long term on the Linux kernel. In Figure 4a, we plot how much time each SAT solver needs for analyzing each feature model. The x-axis forms a timeline of feature models, while the y-axis shows logarithmic SAT runtime in seconds. Each colored line in Figure 4a represents the evolution of one SAT solver. Every point is the median of all measured iterations (cf. $RQ_{1.5}$). We connect points with line segments for improved visualization. We further distinguish formula extraction (cf. $RQ_{1.2}$) and CNF transformation (cf. $RQ_{1.3}$). In Figure 4b, we visualize the same data, but we switch perspectives (i.e., each line represents a feature model). Finally, in Table 4, we perform a regression analysis for the evolution of SAT runtime using Pearson's correlation coefficient r.

Results The runtime of all SAT solvers on all feature models spans roughly 3–4 orders of magnitude, regardless of extractor and CNF transformation. Starting in 2018, the median SAT runtime lies over one second. SAT runtime generally depends on the solver, with recent and early solvers in Figure 4a requiring the least time, and all other solvers requiring more (outliers being 2003's `BerkMin`, 2023's `SBVA-CaDiCaL`, and 2024's `Kissat`). For each individual SAT solver, runtime grows exponentially with a significant and very strong log-linear correlation, slowing down by 13%–29% per year in the median (cf. Table 4). SAT runtime also depends on the feature model, with more recent feature models in Figure 4b generally requiring more time to solve than less recent or the oldest feature models.

Table 4: Linear regression of log10 SAT runtime (RQ_2–RQ_4).

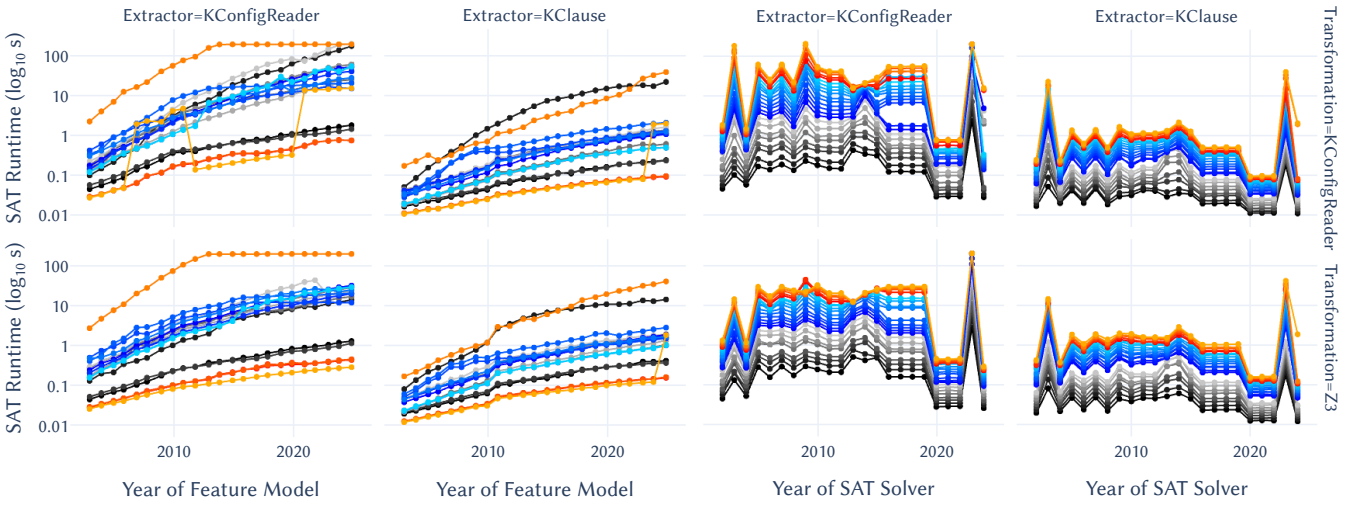
	Solver	p-Value			Effect Size r			Yearly Growth		
	Strategy	Min	Median	Max	Min	Median	Max	Min	Median	Max
SAT Competition	2002	.000	.000	.000	.97	.98	.99	13.6%	16.3%	18.8%
	2003	.000	.000	.000	.94	.98	.99	24.3%	29.4%	41.1%
	2004	.000	.000	.000	.96	.98	.99	12.5%	14.5%	16.2%
	2005	.000	.000	.000	.97	.98	.99	18.0%	21.2%	30.1%
	2006	.000	.000	.000	.95	.97	.99	17.7%	20.4%	28.9%
	2007	.000	.000	.000	.97	.98	.99	17.7%	21.0%	29.8%
	2008	.000	.000	.000	.97	.98	.99	17.0%	22.3%	27.6%
	2009	.000	.000	.000	.94	.98	.99	16.9%	19.9%	37.0%
	2010	.000	.000	.000	.91	.98	.99	18.6%	22.7%	31.8%
	2011	.000	.000	.000	.97	.98	.99	16.9%	20.4%	28.9%
	2012	.000	.000	.000	.97	.98	.99	17.1%	20.5%	29.0%
	2013	.000	.000	.000	.91	.93	.98	13.7%	15.4%	17.4%
	2014	.000	.000	.000	.83	.92	.96	16.3%	17.4%	18.9%
	2015	.000	.000	.000	.89	.97	.98	18.1%	19.7%	22.5%
	2016	.000	.000	.000	.94	.98	.99	16.7%	22.4%	34.1%
	2017	.000	.000	.000	.94	.98	.99	16.7%	22.4%	34.0%
	2018	.000	.000	.000	.94	.98	.99	16.6%	22.4%	34.3%
	2019	.000	.000	.000	.94	.98	.99	16.6%	22.5%	34.4%
	2020	.000	.000	.000	.97	.98	.99	10.7%	13.4%	17.7%
	2021	.000	.000	.000	.97	.98	.99	10.7%	13.4%	17.7%
	2022	.000	.000	.000	.97	.98	.99	10.7%	13.4%	17.6%
	2023	.000	.000	.000	.84	.93	1.00	19.0%	24.3%	29.9%
	2024	.000	.000	*.029*	.46	.88	.99	11.1%	13.5%	24.4%
FIDE	09-11	.000	.000	.000	.96	.99	1.00	5.2%	10.6%	24.1%
	11-14	.000	.000	.000	.96	.99	1.00	5.2%	10.5%	24.2%
	≥ 14	.000	.000	.000	.96	.99	1.00	5.1%	11.0%	25.2%
	Current	.000	.003	*.039*	.43	.61	.67	13.1%	15.9%	22.0%
	Optimal	.000	.000	.000	.81	.84	.87	9.0%	10.4%	13.5%
	Historic	.000	.000	.000	.96	.98	.99	10.5%	11.8%	16.2%
	Pragmatic	.000	.000	.000	.91	.98	.99	4.9%	10.6%	19.7%

In each triple of cells, we summarize eight values (i.e., for two extractors, two CNF transformations, and two architectures) by reporting their minimum, median, and maximum. Statistically insignificant results (i.e., $p > .005$) are *emphasized.*

Discussion The large spread in SAT runtime indicates that the chosen feature model and solver crucially influence the analysis performance on the Linux kernel. In particular, SAT runtime grows over time for any given solver, which is in line with Linux becoming more complex (cf. Figure 1a) [81]. However, the extent of this growth is astonishing, as SAT runtime doubles every 3–6 years (depending on the solver). Thus, SAT solvers age considerably on the Linux kernel.

Notably, on recent feature models, SAT runtime is often higher than one second. This may sound affordable, but it severely hampers repeated SAT queries often encountered in practice [71, 113, 133], which all rely on the lower-bound SAT query evaluated here.

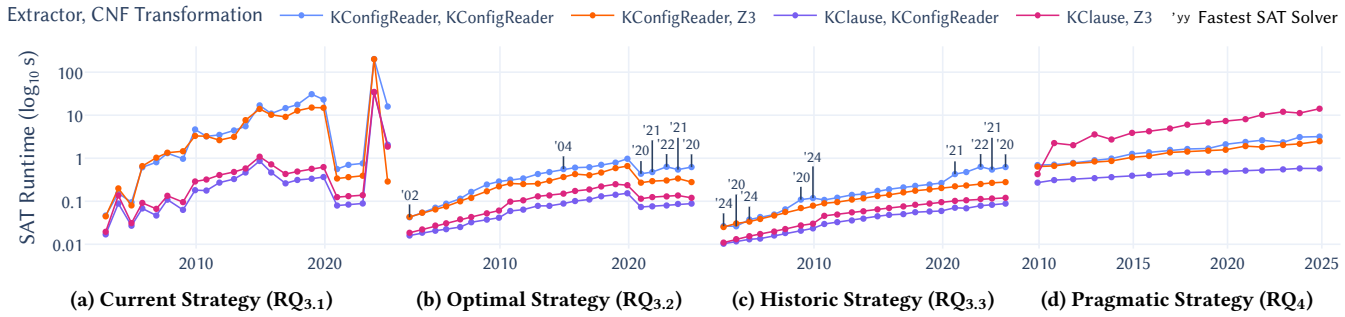
We briefly discuss solver-specific observations. First, some of the earliest solvers (i.e., 2002's `Limmat` and 2004's `zChaff` [90]) are surprisingly efficient. They are only surpassed by the most recent family of `Kissat` [18] solvers from 2020 to 2022. `Limmat` and `zChaff` perform only rudimentary preprocessing, while `Kissat` is an efficient reimplementation of `CaDiCaL` [18] in C with less inprocessing [19]. We hypothesize that these solvers may perform well because the kernel's feature model does not actually require sophisticated pre- or inprocessing. In between these efficient solvers, there is a plateau phase in Figure 4b from 2005 to 2019, where no SAT solver improves noticeably upon the previous ones. Thus, in this period, SAT solvers have essentially stagnated on the kernel.



(a) SAT-Solver Perspective **(b) Feature-Model Perspective**

Both figures show the same data from two different perspectives. In each, we distinguish two extractors (left + right) and two CNF transformations (top + bottom). Each colored line represents one SAT solver (cf. Figure 4a) or feature model (cf. Figure 4b) (2000s = black to gray, 2010s = dark to light blue, 2020s = red to orange).

Figure 4: How does the runtime of historic SAT solvers evolve over the history of the Linux kernel's feature model (RQ_2)?



(a) Current Strategy ($RQ_{3.1}$) **(b) Optimal Strategy ($RQ_{3.2}$)** **(c) Historic Strategy ($RQ_{3.3}$)** **(d) Pragmatic Strategy (RQ_4)**

Figure 5: How does SAT runtime evolve over the Linux kernel's history for different constructed meta-SAT solvers (RQ_3–RQ_4)?

There are three outlier solvers: First, `BerkMin`, which we have only limited information about, so we disregard it. Second, `SBVA-CaDiCaL` is the worst-performing solver in our evaluation due to its SBVA preprocessing [20, 55]. This preprocessing is hard-coded to conclude after 120 seconds on recent feature models, after which `CaDiCaL` easily solves the formula. This demonstrates that more recent solvers are not necessarily more efficient, and that not every innovation is beneficial for Linux. Finally, `Kissat` from 2024 [17] is the most recent winning solver from the SAT competition. However, its performance on Linux is much less predictable on varying versions than its `Kissat` predecessors'. That is, a given feature model is either very hard or very easy to solve for this version of `Kissat`, which leads to huge "jumps" in Figure 4a. On closer inspection, the position of these jumps seem to depend upon the formula's clause order (cf. $RQ_{1.5}$). We hypothesize that these jumps may be due to the problematic SBVA preprocessing [55] in 2024's `Kissat` [17].

5.3.7 $RQ_{3.1}$: Current Strategy. Based on the same data as in RQ_2, we construct meta-SAT solvers according to three strategies in RQ_3. In $RQ_{3.1}$, we investigate how SAT runtime evolves when we analyze the kernel's feature model from year y with the single best solver from the same year y. That is, we try to match each feature model with the leading solver available at the time (cf. Figure 5a). This reflects the typical assumption that the most recent solver is also the fastest overall, which may or may not be true for Linux. If SAT solvers can indeed keep up with the kernel's growth, we expect SAT runtime to stay nearly constant on average here, or even reduce over time.

Results SAT runtime grows exponentially with a significant and moderate to strong log-linear correlation, slowing down by 16% per year in the median (cf. Table 4). The `Kissat` family of solvers from 2020 to 2022 reduces SAT runtime by about one order of magnitude.

Discussion Our results clearly indicate that the most recent SAT solver cannot keep up with Linux, as there is an upward trend. Disruptive breakthroughs like `Kissat` are promising, but cannot completely counteract the kernel's growth. Indeed, there is a slowdown in each solver family (i.e., `Glucose`, `Lingeling` [16], `MapleSAT` [18], and `Kissat`), although each family member attempts to improve upon its predecessors. For the outliers in 2023 and 2024, see RQ_2.

5.3.8 $RQ_{3.2}$: Optimal Strategy. In $RQ_{3.2}$, we slightly modify the strategy from $RQ_{3.1}$: We analyze the feature model from year y with the *fastest* solver on Linux available in year y (i.e., the *virtual best solver* [130]). This corresponds to the idea behind portfolio solvers [131]. In contrast to $RQ_{3.1}$, this strategy does not assume that the most recent solver is also the fastest. However, it is also less common in practice, as it requires running all SAT solvers to determine the fastest one.

Results Analogous to $RQ_{3.1}$, SAT runtime slows down by 10% per year. Again, `Kissat` noticeably reduces SAT runtime (cf. Figure 5b).

Discussion When the optimal SAT solver is known, solving is up to one order of magnitude faster than in $RQ_{3.1}$, so choosing the solver carefully does pay off. However, even the optimal SAT solver still cannot keep up with Linux, as there is a clear upward trend.

5.3.9 $RQ_{3.3}$: Historic Strategy. In $RQ_{3.3}$, we analyze every feature model with the fastest solver available today. This strategy no longer represents "traveling back in time." Instead, it analyzes the kernel retrospectively, as needed for evolutionary analyses [76, 97, 121].

Results SAT runtime slows down by 12% per year. Each solver in the `Kissat` family is the fastest on some models (cf. Figure 5c).

Discussion Clearly, `Kissat` is currently the state of the art for analyzing (any version of) the feature model of the Linux kernel. Thus, researchers should rely on `Kissat` if performance is key.[17]

5.3.10 RQ_4: Pragmatic Strategy. In $RQ_{2–3}$, we cover the academic point of view by investigating historic solvers from the SAT competition (i.e., experiment ①). With RQ_4, we address the industrial point of view (i.e., experiment ②). That is, we study how SAT runtime evolves in FeatureIDE [91], a widely-used product-line tool (cf. Section 4). To this end, we pair each feature model with the version of `Sat4j` [82] that was available in FeatureIDE at the time.

Results `Sat4j` is often faster than solvers from the 2010s and always slower than the `Kissat` solvers (cf. Figure 4a and Figure 5d). It slows down by 11% per year in the median (cf. Table 4). When combined with KClause for extraction and KConfigReader for CNF transformation, `Sat4j` only slows down by 5% per year.

Discussion Surprisingly, `Sat4j` performs quite well on the Linux kernel, although it is Java-based and never ranked highly in the SAT competition. These results mirror RQ_2, where the less sophisticated solvers were also surprisingly efficient. Notably, `Sat4j` has a minimal slowdown of 5% per year, which is much lower than the 9% minimal slowdown of the optimal solver (cf. Figure 5b). Thus, if this trend continues, `Sat4j` may theoretically even outperform the optimal solver at some point. Overall, FeatureIDE may have missed out in raw SAT performance by not exchanging solvers regularly. Still, sticking to `Sat4j` for fifteen years turned out to be remarkably future-proof, performance-wise, and minimized maintenance costs.

5.4 Threats to Validity

We briefly discuss potential threats to our evaluation's validity [129].

Internal Validity We carefully designed our evaluation to identify potentially confounding variables (cf. Table 1), and either eliminate or evaluate them (cf. RQ_1). As for tooling-related threats, we refer to the `README` file of torte [78],[6] where we share many details.

External Validity: Subject Systems We focus on the Linux kernel as a relevant case study (cf. Section 3). This limits the generalizability of our results to other subject systems. However, it is known that small open-source [117] and large industrial [8, 115] systems grow in configurability over time as well. It is plausible that SAT solvers may also have trouble keeping up with these systems, depending on their growth and the complexity of their feature dependencies. Moreover, we can also not know for sure how the kernel or SAT solvers will develop in the future. However, given the longitude of our study, we can extrapolate that analyses on Linux will become increasingly harder if no countermeasures are taken.

External Validity: Solvers and Analyses We focus on solvers that have been awarded in the SAT competition, and we only perform a single SAT query. While this is well-justified in the context of our systematic, large-scale evaluation (cf. Section 4), non-winning solvers and more complex analysis queries may potentially yield additional relevant insights. To investigate this, we conduct a third, smaller-scale[18] experiment with a different set of SAT solvers and analysis queries. For SAT solving, we randomly choose two solvers that are *not* winners from each SAT competition in the years 2002, '05, '08, '11, '14, '17, '20, and '23.[19] This yields 16 non-winning SAT solvers in total, including several local-search and parallel solvers. For analysis, we evaluate two additional SAT query types that scale on Linux and often occur in practice [113, 114]: the consistency of features (e.g., needed for core and dead features [9, 21, 64]) and partial configurations (e.g., needed for atomic sets [57, 106] and sampling [125]).[20] In total, this yields roughly 100,000 additional SAT queries that we execute (requiring about five days of solving time). In the following, we only give a brief summary; for transparency, we make the full experiment description and results (including diagrams) available in our reproduction package.[6]

Overall, the results of this additional experiment confirm our main findings from $RQ_{2–4}$. As expected, the non-winning solvers are typically slower across all queries (by factor 2.94 in the median) than the SAT competition's winning solvers, and they also age considerably: For a simple SAT query, the non-winning solvers slow down by 4%–34% per year (16.2% in the median), as well as 4% and 5% in the optimal and historic strategies, respectively. In particular, no non-winning solver outperforms `Kissat` from 2020. Nonetheless, we identify two promising non-winning solvers (i.e., `BlackHoleSAT` from 2011 and `Tinisat` [59] from 2008) that are only 3.2 and 3.6 times slower than `Kissat` for a simple SAT query on the latest kernel version, and only slow down 4.2% and 4.9% per year, respectively. Thus, these solvers have potential to outperform `Kissat` in the future, similar to `Sat4j` (cf. RQ_4). Both `BlackHoleSAT` and `Tinisat` have very simple implementations (1100 and 600 source lines of code, respectively), which shows that simple solvers can compete surprisingly well with complex ones.

[17] We recommend to use `Kissat` from before 2024, which are more predictable (cf. RQ_2).

[18] We significantly reduce the scope of this preliminary experiment by only evaluating one architecture (the widely used `x86`), one extractor (KClause, which has smaller formulas), one CNF transformation (the industrial-grade Z3), and one iteration.

[19] We only evaluate solvers in intervals of three years to keep execution time manageable. We exclude defunct solvers and solvers that are related to a winning solver (e.g., `Kissat-*`). As a baseline, we evaluate the winning solvers from the respective years.

[20] For feature consistency, we evaluate the two queries $\mathrm{SAT}(\mathrm{FM} \wedge f)$ and $\mathrm{SAT}(\mathrm{FM} \wedge \neg f)$ on 50 random (but fixed) features f per feature model. For partial configuration consistency, we evaluate the four queries $\mathrm{SAT}(\mathrm{FM} \wedge (\neg)f \wedge (\neg)g)$ on 25 random pairs of features f and g. For the sake of completeness, we also evaluate $\mathrm{SAT}(\mathrm{FM})$.

Table 5: Summary of key insights from our evaluation.

RQ	Key Insight
$RQ_{1.1}$	The architectures `x86` and `arm` are of similar computational complexity.
$RQ_{1.2}$	Different encodings of `tristate` features noticeably affect SAT runtime.
$RQ_{1.3}$	Different CNF transformations affect SAT runtime, but only slightly.
$RQ_{1.4}$	Different compilers of SAT solver binaries do not affect SAT runtime.
$RQ_{1.5}$	Nondeterministic extraction and CNF transformation affect SAT runtime.
RQ_2	Every year, historic SAT solvers slow down by 13%–29%.
RQ_3	Historic SAT solvers cannot keep up with the Linux kernel's feature model.
RQ_4	A pragmatic SAT solver cannot either, slowing down by 11% per year.

Our results for the additional analysis queries confirm these observations without major deviations, which suggests that these types of queries are not fundamentally harder or easier to solve on Linux than simple SAT queries. Overall, this additional experiment strengthens our confidence in the generalizability of our main results to other solvers and analysis queries on Linux.

5.5 Insights and Recommendations

To conclude our evaluation, we summarize our key insights in Table 5. Overall, SAT solvers aged considerably on the Linux kernel over the last two decades, and they cannot keep up with its growth. In fact, we expect optimal SAT runtime to double every seven years (cf. $RQ_{3.2}$). So, it is even unclear whether advanced analyses will scale to Linux in 10–20 years. This establishes the Linux kernel as a cautionary tale for researchers and practitioners.

So, can we improve this situation in the future? One solution may be to trust in continued hardware improvements (i.e., Moore's law [105]) to offset the slowdown [42]. However, it is questionable how long this law will hold [36]. We cannot trust in better compilers, either (cf. $RQ_{1.4}$). Instead, we envision the following angles of action:

- First, kernel developers can aim to reduce variability, or at least limit its growth more [81]. This kind of variability reduction is often called for, but rarely practiced consequently [5, 8, 117].
- Second, automated reasoning researchers can further improve SAT solvers, as they have the expert knowledge needed to develop better algorithms. `Kissat` exemplifies (cf. $RQ_{3.2}$) how such disruptive innovations can temporarily offset the yearly slowdown. Moreover, SAT researchers can consider including Linux and other product lines in their competition benchmarks.
- Third, software engineering researchers can make their product-line tools comply with standardized interfaces (e.g., IPASIR [11]). This would allow for seamless upgrades and exchanges of SAT solvers. After such an upgrade, they can check for performance regressions (cf. $RQ_{3.1}$). Also, they can create compositional analyses [2] to cut the feature model into smaller, simpler pieces. Finally, more efficient knowledge compilation [113, 119] can mitigate performance loss due to repeated SAT queries [133].

These stakeholders can even collaborate to tackle the slowdown [45].

6 Related Work

The evolution and growth of configurable software has been studied on various product lines [33, 76, 115], including Linux [35, 52, 61, 89, 104]. Individual evolution steps have also been considered in incremental SAT solving [11, 38, 75] and evolutionary analyses [4, 76, 97, 121]. However, we are not aware of any study that investigates the long-term evolution of propositional formulas.

It is well-known that SAT solving performance has improved remarkably over the years [43]. Recently, Biere et al. [19] initiated the SAT museum,[11] in which they collect, patch, and evaluate solvers from the SAT competition. While they fix instances and vary solvers, we vary both (solvers and instances) to determine whether SAT solvers can keep up with Linux. Moreover, we study whether the compiler is a confounding variable, which they hypothesize about.

Complementing the SAT museum's black-box evaluation, several studies have attempted to isolate key elements that make SAT solving successful [44, 72, 75]. These studies attempt to "dissect" how a single configurable SAT solver performs on a fixed set of instances. Our work is orthogonal, as we evaluate a collection of unmodified SAT solvers on a set of evolving formulas.

Fichte et al. [42] evaluate recent SAT solvers on old hardware and old SAT solvers on recent hardware. Thus, they investigate whether hardware or algorithmic improvements have greater impact on SAT solving performance, testing a variation of Wirth's law [128]. We add another lens to this discussion by evaluating recent SAT solvers on old feature models and old SAT solvers on recent feature models.

7 Conclusion

The Linux kernel is growing more configurable. (Semi-)automated analyses support kernel developers in reasoning about its increasingly complex set of feature dependencies. These analyses are often computed with SAT solvers, which keep being improved by the SAT community. In this paper, we find that despite all improvements, SAT solvers cannot keep up with the Linux kernel's feature model. If this trend continues, Linux will become increasingly difficult to analyze. To prevent this, we make several recommendations to kernel developers and researchers. Moreover, we raise awareness that propositional formulas derived from product lines evolve over time, which has rarely been addressed before in SAT research.

In future work, we aim to generalize our evaluation to other subject systems, solver classes, and analyses. While this is methodically challenging (e.g., due to the current lack of published feature-model histories), it may be possible and worthwhile to assemble a bigger picture from numerous smaller puzzle pieces. In particular, we aim to extend TORTE [78] so we can extract and evaluate feature-model histories for other (smaller) systems, such as BusyBox [127], uClibc, and uClibc-ng [15]. Altogether, these systems cover a similar time frame as the Linux kernel, which makes them suitable candidates for future studies. Due to the smaller size of these systems, we can then also efficiently study computationally demanding solver classes (e.g., local search [58] or #SAT [41] solvers). Finally, we aim to study more complex analyses (e.g., core and dead features [21, 64] and atomic sets [57, 106]), and how the kernel's growth affects the number of repeated SAT queries needed by such analyses [113].

Acknowledgments

We would like to thank Paul Maximilian Bittner, Raphael Dunkel, Thomas Leich for discussions on this work. We also thank the anonymous reviewers for their constructive and helpful feedback.

References

[1] I. Abal, C. Brabrand, and A. Wąsowski. 2014. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *ASE*. ACM, 421–432.

[2] M. Acher, P. Collet, P. Lahire, and R. B. France. 2011. Slicing Feature Models. In *ASE*. IEEE, 424–427.

[3] M. Acher, B. Combemale, G. A. Randrianaina, and J.-M. Jezequel. 2024. Embracing Deep Variability For Reproducibility and Replicability. In *REP*. ACM.

[4] M. Acher, P. Heymans, P. Collet, C. Quinton, P. Lahire, and P. Merle. 2012. Feature Model Differences. In *CAiSE*. Springer, 629–645.

[5] M. Acher, L. Lesoil, G. A. Randrianaina, X. Tërnava, and O. Zendra. 2023. A Call for Removing Variability. In *VaMoS*. ACM, 82–84.

[6] M. Acher, H. Martin, L. Lesoil, A. Blouin, J.-M. Jézéquel, D. E. Khelladi, O. Barais, and J. A. Pereira. 2022. Feature Subset Selection for Learning Huge Configuration Spaces: The Case of Linux Kernel Size. In *SPLC*. ACM, 85–96.

[7] B. Adams, K. D. Schutter, H. Tromp, and W. de Meuter. 2008. The Evolution of the Linux Build System. *ECEASST* 8 (2008).

[8] V. Antinyan. 2025. Pursuit of Automotive Software Variant Reduction for Engineering Simplicity. In *FSE Companion*. ACM, 87–92.

[9] S. Apel, D. Batory, C. Kästner, and G. Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.

[10] G. Audemard, L. Paulevé, and L. Simon. 2020. SAT Heritage: A Community-Driven Effort for Archiving, Building and Running More Than Thousand SAT Solvers. In *SAT*. Springer, 107–113.

[11] T. Balyo, A. Biere, A. Iser, and C. Sinz. 2016. SAT Race 2015. *AIJ* 241 (2016).

[12] D. Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *SPLC*. Springer, 7–20.

[13] D. Benavides, S. Segura, and A. Ruiz-Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–708.

[14] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *VaMoS*. ACM, 7:1–7:8.

[15] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *TSE* 39, 12 (2013), 1611–1640.

[16] A. Biere. 2014. Lingeling Essentials, A Tutorial on Design and Implementation Aspects of the the SAT Solver Lingeling. In *POS*, Daniel Le Berre (Ed.), Vol. 27. EasyChair, 88.

[17] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt. 2024. *CaDiCaL, Gimsatul, IsaSAT and Kissat Entering the SAT Competition 2024*. Technical Report Report B-2024-1. University of Helsinki.

[18] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger. 2020. *CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020*. Technical Report Report B-2020-1. University of Helsinki.

[19] A. Biere, M. Fleury, N. Froleyks, and M. J. H. Heule. 2023. The SAT Museum. In *POS*. 72–87.

[20] A. Biere, M. Fleury, and F. Pollitt. 2023. *CaDiCaL_vivinst, IsaSAT, Gimsatul, Kissat, and TabularaSAT Entering the SAT Competition 2023*. Technical Report Report B-2023-1. University of Helsinki. 14–15 pages.

[21] A. Biere, N. Froleyks, and W. Wang. 2023. CadiBack: Extracting Backbones with CaDiCaL. In *SAT (LIPIcs, Vol. 271)*, Meena Mahajan and Friedrich Slivovsky (Eds.). Schloss Dagstuhl, 3:1–3:12.

[22] A. Biere and M. Heule. 2019. The Effect of Scrambling CNFs. In *POS*, Vol. 59.

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

[24] H. K. Büning and T. Lettmann. 1999. *Propositional Logic: Deduction and Algorithms*. Vol. 48. Cambridge University Press.

[25] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. 2003. Feature Interaction: A Critical Review and Considered Forecast. *ComNet* 41, 1 (2003), 115–141.

[26] T. Castro, L. Teixeira, V. Alves, S. Apel, M. Cordy, and R. Gheyi. 2021. A Formal Framework of Software Product Line Analyses. *TOSEM* 30, 3, Article 34 (2021).

[27] K. Claessen, N. Een, M. Sheeran, and N. Sorensson. 2008. SAT-Solving in Practice. In *WODES*. IEEE, 61–67.

[28] P. C. Clements, J. D. McGregor, and S. G. Cohen. 2005. *The Structured Intuitive Model for Product Line Economics (SIMPLE)*. Technical Report.

[29] 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.

[30] M. Davis, G. Logemann, and D. Loveland. 1962. A Machine Program for Theorem-Proving. *Comm. ACM* 5, 7 (1962), 394–397.

[31] M. Davis and H. Putnam. 1960. A Computing Procedure for Quantification Theory. *J. ACM* 7, 3 (1960), 201–215.

[32] L. De Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. Springer, 337–340.

[33] R. P. de Oliveira, A. R. Santos, E. S. de Almeida, and G. S. da Silva Gomes. 2017. Evaluating Lehman's Laws of Software Evolution Within Software Product Lines Industrial Projects. *JSS* 131 (2017), 347–365.

[34] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. 2012. Understanding Linux Feature Distribution. In *MISS*. ACM, 15–20.

[35] N. Dintzner, A. van Deursen, and M. Pinzger. 2017. Analysing the Linux Kernel Feature Model Changes Using FMDiff. *SoSyM* 16 (2017), 55–76.

[36] L. Eeckhout. 2017. Is Moore's Law Slowing Down? What's Next? *IEEE Micro* 37, 4 (2017), 4–5.

[37] S. El-Sharkawy, A. Krafczyk, and K. Schmid. 2015. Analysing the KConfig Semantics and its Analysis Tools. In *GPCE*. ACM, 45–54.

[38] K. Fazekas, A. Niemetz, M. Preiner, M. Kirchweger, S. Szeider, and A. Biere. 2023. IPASIR-UP: User Propagators for CDCL. In *SAT (LIPIcs, Vol. 271)*. Schloss Dagstuhl, 8:1–8:13.

[39] W. Fenske, J. Meinicke, S. Schulze, S. Schulze, and G. Saake. 2017. Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line. In *SANER*. IEEE, 316–326.

[40] D. Fernandez-Amoros, R. Heradio, C. Mayr-Dorn, and A. Egyed. 2019. A KConfig Translation to Logic With One-Way Validation System. In *SPLC*. ACM, 303–308.

[41] J. K. Fichte, M. Hecher, and F. Hamiti. 2021. The Model Counting Competition 2020. *JEA* 26, Article 13 (2021).

[42] J. K. Fichte, M. Hecher, and S. Szeider. 2020. A Time Leap Challenge for SAT-Solving. In *CP*, Helmut Simonis (Ed.). Springer, 267–285.

[43] J. K. Fichte, D. Le Berre, M. Hecher, and S. Szeider. 2023. The Silent (R)evolution of SAT. *Comm. ACM* 66, 6 (2023), 64–72.

[44] M. Fleury and D. Kaufmann. 2023. Life Span of SAT Techniques. In *POS*.

[45] P. Franz, T. Berger, I. Fayaz, S. Nadi, and E. Groshev. 2021. ConfigFix: Interactive Configuration Conflict Resolution for the Linux Kernel. In *ICSE-SEIP*. IEEE, 91–100.

[46] C. Fritsch, R. Abt, and B. Renz. 2020. The Benefits of a Feature Model in Banking. In *SPLC*. ACM, Article 9.

[47] N. Froleyks, M. Heule, A. Iser, M. Järvisalo, and M. Suda. 2021. SAT Competition 2020. *AIJ* 301 (2021), 103572.

[48] J. A. Galindo, M. Acher, J. M. Tirado, C. Vidal, B. Baudry, and D. Benavides. 2016. Exploiting the Enumeration of All Feature Model Configurations: A New Perspective With Distributed Computing. In *SPLC*. ACM, 74–78.

[49] J. A. Galindo, D. Benavides, P. Trinidad, A.-M. Gutiérrez-Fernández, and A. Ruiz-Cortés. 2019. Automated Analysis of Feature Models: Quo Vadis? *Computing* 101, 5 (2019), 387–433.

[50] J. A. Galindo, J. M. Horcas, A. Felfernig, D. Fernández-Amorós, and D. Benavides. 2023. FLAMA: A Collaborative Effort to Build a New Framework for the Automated Analysis of Feature Models. In *SPLC*. ACM, 16–19.

[51] T. Glatard, L. B. Lewis, R. Ferreira da Silva, R. Adalat, N. Beck, C. Lepage, P. Rioux, M.-E. Rousseau, T. Sherif, E. Deelman, N. Khalili-Mahani, and A. C. Evans. 2015. Reproducibility of Neuroimaging Analyses Across Operating Systems. *FNINF* 9 (2015).

[52] M. Godfrey and Q. Tu. 2000. Evolution in Open Source Software: A Case Study. In *ICSM*. 131–142.

[53] M. W. Godfrey and D. M. German. 2014. On the Evolution of Lehman's Laws. *JSEP* 26, 7 (2014), 613–619.

[54] K. Gupta and T. Sharma. 2021. Changing Trends in Computer Architecture: A Comprehensive Analysis of ARM and x86 Processors. *IJSRCSEIT* 7, 3 (2021), 619–631.

[55] A. Haberlandt, H. Green, and M. J. H. Heule. 2023. Effective Auxiliary Variables via Structured Reencoding. In *SAT (LIPIcs, Vol. 271)*. Schloss Dagstuhl, 11:1–11:19.

[56] M. Hentze, T. Pett, T. Thüm, and I. Schaefer. 2021. Hyper Explanations for Feature-Model Defect Analysis. In *VaMoS*. ACM, Article 14.

[57] T. Heß and A. Molt. 2025. *A Fast Counting-Free Algorithm for Computing Atomic Sets in Feature Models*. Technical Report arXiv:2501.12490.

[58] H. H. Hoos and T. Stützle. 2000. Local Search Algorithms for SAT: An Empirical Evaluation. *JAR* 24, 4 (2000), 421–481.

[59] J. Huang. 2007. A Case for Simple SAT Solvers. In *CP*. Springer, 839–846.

[60] A. Iser, C. Sinz, and M. Taghdiri. 2013. Minimizing Models for Tseitin-Encoded SAT Instances. In *SAT*. Springer, 224–232.

[61] A. Israeli and D. G. Feitelson. 2010. The Linux Kernel as a Case Study in Software Evolution. *JSS* 83, 3 (2010), 485–501.

[62] P. Jackson and D. Sheridan. 2004. Clause Form Conversions for Boolean Circuits. In *SAT*. Springer, 183–198.

[63] M. Janota. 2008. Do SAT Solvers Make Good Configurators?. In *SPLC*, Vol. 2. University of Limerick, Lero, 191–195.

[64] M. Janota, I. Lynce, and J. Marques-Silva. 2015. Algorithms for Computing Backbones of Propositional Formulae. *AIC* 28, 2 (2015), 161–177.

[65] M. Järvisalo, A. Biere, and M. Heule. 2010. Blocked Clause Elimination. In *TACAS*. Springer, 129–144.

[66] M. Järvisalo, D. Le Berre, O. Roussel, and L. Simon. 2012. The International SAT Solver Competitions. *AI Magazine* 33, 1 (2012), 89–92.

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

[68] S. R. Kamali, S. Kasaei, and R. E. Lopez-Herrejon. 2019. Answering the Call of the Wild? Thoughts on the Elusive Quest for Ecological Validity in Variability Modeling. In *MODEVAR*. ACM, 143–150.

[69] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Software Engineering Institute.

[70] C. Kästner. 2017. *Differential Testing for Variational Analyses: Experience From Developing KConfigReader*. Technical Report arXiv:1706.09357.

[71] C. Kästner, K. Ostermann, and S. Erdweg. 2012. A Variability-Aware Module System. In *OOPSLA*. ACM, 773–792.

[72] H. Katebi, K. A. Sakallah, and J. P. Marques-Silva. 2011. Empirical Study of the Anatomy of Modern SAT Solvers. In *SAT*. Springer, 343–356.

[73] O. Kautz. 2023. The Complexities of the Satisfiability Checking Problems of Feature Diagram Sublanguages. *SoSyM* 22, 4 (2023), 1113–1129.

[74] A. Knüppel, T. Thüm, S. Mennicke, J. Meinicke, and I. Schaefer. 2017. Is There a Mismatch Between Real-World Feature Models and Product-Line Research?. In *ESEC/FSE*. ACM, 291–302.

[75] S. Kochemazov, A. Ignatiev, and J. Marques-Silva. 2021. Assessing Progress in SAT Solvers Through the Lens of Incremental SAT. In *SAT*. Springer, 280–298.

[76] S. Krieter, R. Arens, M. Nieke, C. Sundermann, T. Heß, T. Thüm, and C. Seidl. 2021. Incremental Construction of Modal Implication Graphs for Evolving Feature Models. In *SPLC*. ACM, 64–74.

[77] G. Kroah-Hartman. 2003. The Kernel Configuration and Build Process. *Linux* 2003, 109 (2003).

[78] E. Kuiter. 2026. torte: Reproducible Feature-Model Experiments à la Carte. In *ICSE*. ACM, New York, NY, USA. To appear.

[79] E. Kuiter, T. Heß, C. Sundermann, S. Krieter, T. Thüm, and G. Saake. 2024. How Easy is SAT-Based Analysis of a Feature Model?. In *VaMoS*. ACM, 149–151.

[80] E. Kuiter, S. Krieter, C. Sundermann, T. Thüm, and G. Saake. 2022. Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. In *ASE*. ACM.

[81] E. Kuiter, C. Sundermann, T. Thüm, T. Heß, S. Krieter, and G. Saake. 2025. How Configurable is the Linux Kernel? Analyzing Two Decades of Feature-Model History. *TOSEM* (2025). To appear.

[82] D. Le Berre and A. Parrain. 2010. The Sat4j Library, Release 2.2. *JSAT* 7, 2-3 (2010), 59–64.

[83] M. M. Lehman. 1996. Laws of Software Evolution Revisited. In *EWSPT*. Springer, 108–124.

[84] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. 1997. Metrics and Laws of Software Evolution–The Nineties View. In *METRICS*. IEEE.

[85] L. Lesoil, M. Acher, A. Blouin, and J.-M. Jézéquel. 2021. Deep Software Variability: Towards Handling Cross-Layer Configuration. In *VaMoS*. ACM, 10:1–10:8.

[86] J. H. Liang, V. Ganesh, K. Czarnecki, and V. Raman. 2015. SAT-Based Analysis of Large Real-World Feature Models Is Easy. In *SPLC*. Springer, 91–100.

[87] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. 2013. Scalable Analysis of Variable Software. In *ESEC/FSE*. ACM, 81–91.

[88] M. H. Liffiton and K. A. Sakallah. 2008. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *JAR* 40, 1 (2008), 1–33.

[89] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski. 2010. Evolution of the Linux Kernel Variability Model. In *SPLC*. Springer, 136–150.

[90] Y. S. Mahajan, Z. Fu, and S. Malik. 2004. Zchaff2004: An Efficient SAT Solver. In *SAT (LNCS, Vol. 3542)*. Springer, 360–375.

[91] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake. 2017. *Mastering Software Variability With FeatureIDE*. Springer.

[92] M. Mendonça, A. Wąsowski, and K. Czarnecki. 2009. SAT-Based Analysis of Feature Models Is Easy. In *SPLC*. Software Engineering Institute, 231–240.

[93] J. Mortara and P. Collet. 2021. Capturing the Diversity of Analyses on the Linux Kernel Variability. In *SPLC*. ACM, 160–171.

[94] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. 2001. Chaff: Engineering an Efficient SAT Solver. In *DAC*. ACM, 530–535.

[95] L. Murphy, M. Saifi, A. Di Sandro, and M. Chechik. 2025. A Structural Taxonomy for Lifted Software Product Line Analyses. *JSS* 222 (2025), 112280.

[96] S. Nadi, C. Dietrich, R. Tartler, R. C. Holt, and D. Lohmann. 2013. Linux Variability Anomalies: What Causes Them and How Do They Get Fixed?. In *MSR*. IEEE, 111–120.

[97] M. Nieke, J. Mauro, C. Seidl, T. Thüm, I. C. Yu, and F. Franzke. 2018. Anomaly Analyses for Feature-Model Evolution. In *GPCE*. ACM, 188–201.

[98] J. Oh, P. Gazzillo, D. Batory, M. Heule, and M. Myers. 2019. *Uniform Sampling From Kconfig Feature Models*. Technical Report TR-19-02. University of Texas at Austin, Department of Computer Science.

[99] J. Oh, N. F. Yıldıran, J. Braha, and P. Gazzillo. 2021. Finding Broken Linux Configuration Specifications by Statically Analyzing the Kconfig Language. In *ESEC/FSE*. ACM, 893–905.

[100] L. Passos, M. Novakovic, Y. Xiong, T. Berger, K. Czarnecki, and A. Wąsowski. 2011. A Study of Non-Boolean Constraints in Variability Models of an Embedded Operating System. In *FOSD (SPLC)*. ACM, Article 2.

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

[102] Q. Plazar, M. Acher, G. Perrouin, X. Devroey, and M. Cordy. 2019. Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet?. In *ICST*. IEEE, 240–251.

[103] J. Rosenberg. 1997. Some Misconceptions About Lines of Code. In *METRICS*. IEEE, 137–142.

[104] V. Rothberg, N. Dintzner, A. Ziegler, and D. Lohmann. 2016. Feature Models in Linux: From Symbols to Semantics. In *VaMoS*. ACM, 65–72.

[105] R. R. Schaller. 1997. Moore's Law: Past, Present and Future. *IEEE Spectrum* 34, 6 (1997), 52–59.

[106] S. Segura. 2008. Automated Analysis of Feature Models Using Atomic Sets. In *SPLC*, Vol. 2. IEEE, 201–207.

[107] S. She and T. Berger. 2010. *Formal Semantics of the Kconfig Language*. Technical Report. University of Waterloo.

[108] J. P. M. Silva and K. A. Sakallah. 2003. GRASP - A New Search Algorithm for Satisfiability. In *The Best of ICCAD*. Springer, 73–89.

[109] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. 2007. Is the Linux Kernel a Software Product Line?. In *OSSPL*. IEEE, 9–12.

[110] I. Skoulis, P. Vassiliadis, and A. Zarras. 2014. Open-Source Databases: Within, Outside, Or Beyond Lehman's Laws of Software Evolution?. In *CAiSE*. Springer, 379–393.

[111] C. Sundermann, V. F. Brancaccio, E. Kuiter, S. Krieter, T. Heß, and T. Thüm. 2024. Collecting Feature Models from the Literature: A Comprehensive Dataset for Benchmarking. In *SPLC*. ACM, 54–65.

[112] C. Sundermann, T. Heß, M. Nieke, P. M. Bittner, J. M. Young, T. Thüm, and I. Schaefer. 2023. Evaluating State-of-the-Art #SAT Solvers on Industrial Configuration Spaces. *EMSE* 28, 2 (2023), 38.

[113] C. Sundermann, E. Kuiter, T. Heß, H. Raab, S. Krieter, and T. Thüm. 2024. On the Benefits of Knowledge Compilation for Feature-Model Analyses. *AMAI* 92, 5 (2024), 1013–1050.

[114] C. Sundermann, M. Nieke, P. M. Bittner, T. Heß, T. Thüm, and I. Schaefer. 2021. Applications of #SAT Solvers on Feature Models. In *VaMoS*. ACM, Article 12.

[115] C. Sundermann, T. Thüm, and I. Schaefer. 2020. Evaluating #SAT Solvers on Industrial Feature Models. In *VaMoS*. ACM, Article 3.

[116] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *EuroSys*. ACM, 47–60.

[117] X. Tërnava, G. A. Randrianaina, L. Lesoil, and M. Acher. 2025. *Small Yet Configurable: Unveiling Null Variability in Software*. Technical Report. HAL. To appear.

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

[119] T. Thüm. 2020. A BDD for Linux? The Knowledge Compilation Challenge for Variability. In *SPLC*. ACM, Article 16.

[120] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *CSUR* 47, 1 (2014), 6:1–6:45.

[121] T. Thüm, D. Batory, and C. Kästner. 2009. Reasoning About Edits to Feature Models. In *ICSE*. IEEE, 254–264.

[122] L. Torvalds. 2019. Linux Operating System. Website: https://www.kernel.org/. Accessed: 2019-12-02.

[123] G. S. Tseitin. 1983. *On the Complexity of Derivation in Propositional Calculus*. Springer, 466–483.

[124] L. G. Valiant. 1979. The Complexity of Computing the Permanent. *TCS* 8, 2 (1979), 189–201.

[125] M. Varshosaz, M. Al-Hajjaji, T. Thüm, T. Runge, M. R. Mousavi, and I. Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. In *SPLC*. ACM, 1–13.

[126] M. Walch, R. Walter, and W. Küchlin. 2015. Formal Analysis of the Linux Kernel Configuration With SAT Solving. In *ConfWS*.

[127] N. Wells. 2000. Busybox: A Swiss Army Knife for Linux. *Linux* 2000, 78es (2000). Website: https://busybox.net/.

[128] N. Wirth. 1995. A Plea for Lean Software. *IEEE Computer* 28, 2 (1995), 64–68.

[129] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell. 2012. *Experimentation in Software Engineering*. Springer.

[130] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. 2012. Evaluating Component Solver Contributions to Portfolio-Based Algorithm Selectors. In *SAT*. Springer, 228–241.

[131] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. 2008. SATzilla: Portfolio-Based Algorithm Selection for SAT. *JAIR* 32 (2008), 565–606.

[132] N. F. Yıldıran, J. Oh, J. Lawall, and P. Gazzillo. 2024. Maximizing Patch Coverage for Testing of Highly-Configurable Software without Exploding Build Times. *PACMSE* 1, FSE (2024).

[133] J. M. Young, P. M. Bittner, E. Walkingshaw, and T. Thüm. 2023. Variational Satisfiability Solving: Efficiently Solving Lots of Related SAT Problems. *EMSE* 28, 1 (2023), 53.

[134] L. Yu and A. Mishra. 2013. An Empirical Study of Lehman's Law on Software Quality Evolution. *Int. J. Softw. Inform.* 7, 3 (2013), 469–481.