# Feature Subset Selection for Learning Huge Configuration Spaces: The case of Linux Kernel Size

Mathieu Acher, Hugo Martin, Juliana Alves Pereira, Luc Lesoil, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi, Olivier Barais

# Feature Subset Selection for Learning Huge Configuration Spaces: The case of Linux Kernel Size

Mathieu Acher, Hugo Martin, Luc Lesoil, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi and Olivier Barais
Univ Rennes, INSA Rennes, CNRS, Inria, IRISA
Rennes, France

Juliana Alves Pereira
PUC-Rio
Rio de Janeiro, Brazil

## ABSTRACT

Linux kernels are used in a wide variety of appliances, many of them having strong requirements on the kernel size due to constraints such as limited memory or instant boot. With more than nine thousands of configuration options to choose from, developers and users of Linux actually spend significant effort to document, understand, and eventually tune (combinations of) options for meeting a kernel size. In this paper, we describe a large-scale endeavour automating this task and predicting a given Linux kernel binary size out of unmeasured configurations. We first experiment that state-of-the-art solutions specifically made for configurable systems such as performance-influence models cannot cope with that number of options, suggesting that software product line techniques may need to be adapted to such huge configuration spaces. We then show that tree-based feature selection can learn a model achieving low prediction errors over a reduced set of options. The resulting model, trained on 95 854 kernel configurations, is fast to compute, simple to interpret and even outperforms the accuracy of learning without feature selection.

## 1 INTRODUCTION

Linux kernels are used in a wide variety of systems, ranging from embedded devices and cloud services to powerful supercomputers [72]. Many of those systems have strong requirements on the kernel size due to constraints such as limited memory or instant boot [26, 53]. The Linux kernel has a huge configuration space: more than 9,000 binary and independent options, that would indeed yield $2^{9000}$ possible variants of the kernel. Of course not all options are independent - leading to fewer possible variants, but some of them have tri-states values: yes, no, or module instead of simply boolean values - leading to more possible variants. Users can thus choose values for activating options – either compiled as modules or directly integrated within the kernel – and deactivate options. The assignment of a specific value to each option forms a *configuration*, from which a kernel can hopefully be compiled, built and booted. A fundamental issue is that configuration options often have a significant influence on non-functional properties (here: binary size) that are hard to estimate and model *a priori*. To efficiently configure a Linux kernel optimizing these non-functional properties, we first need interpretable information, such as the list of most important options and their effect on non-functional properties. Hubaux *et al.* [28] report that many Linux users complain about the lack of guidance for making configuration decisions and the low quality of the advice provided by the configurators. Beyond Linux and even for much smaller configurable systems, similar configuration issues have been reported [5, 27, 62, 75, 77, 78, 80].

The effort of the Linux community to document options related to kernel binary size is highly valuable, but mostly relies on human expertise, which makes the maintenance of this knowledge quite challenging on the long run. Furthermore, numerous works have shown that only quantifying the performance influence of each individual option is not sufficient in most cases [23, 32, 57, 60, 63]. That is, the performance influence of $n$ options, all jointly activated in a configuration, is not easily deducible from the performance influence of each individual option. As our empirical results will show, the Linux kernel binary size is not an exception: options such as `CONFIG_DEBUG_*` or `CC_OPTIMIZE_FOR_SIZE` have cross-cutting, non-linear effects and cannot be reduced to additive effects, hence basic linear regression models, which are unable to capture interactions among options, give poorly accurate results. To overcome this problem, researchers have considered the problem of learning the effects of options and their interactions in different domains [23, 32, 57, 60, 63]. However, the number of options was limited to dozens of options. At the scale of Linux, there are more than 9000 options and only a fraction of possible kernel configurations can be measured owing to the cost of compiling and building kernels. Linux thus questions whether learning methods used in the state of the art would scale (w.r.t. training time), provide accurate models, and interpretable information at such an unprecedented scale. The novelty of our empirical inquiry is twofold. *First*, we aim to predict a non-functional property (the binary size) of the Linux kernel by considering *all 9K+ options without a priori selection based on documentation or expert knowledge*. *Second*, we aim to automate the process of selecting a subset of relevant options that matter when optimizing the size of the Linux kernel and compare it to expert knowledge.

The contributions of this paper are as follows:

- The design and implementation of a large study about kernel sizes. We engineer specific methods for feature encoding, construction and selection;
- A comparison of a wide range of machine learning algorithms and the effects of tree-based feature selection over prediction errors, interpretability, and training time. We find that tree-based feature selection is able to identify a reduced set of options that both speed up the training time and produce simpler and more accurate models;
- A qualitative analysis of identified options based on the cross-analysis of Linux documentation, default option values and configurations, and experts' knowledge. We find that the subset of options found by tree-based feature selection is consistent w.r.t. Linux knowledge;
- An infrastructure, called TuxML, and a comprehensive dataset of 95 854 configurations with learning procedures for reproducibility of our results [69].

Section 2 introduces the domain knowledge related to the Linux kernel. Section 3 and Section 4 present our research questions and experimental protocol. Section 5 evaluates this protocol on the Linux kernel and summarises key results. Section 6 mentions threats to validity that could affect our experiment. Section 7 lists some related work. Section 8 concludes our paper.

## 2 SIZE MATTERS

This section motivates our work and provides domain knowledge on the Linux kernel.

### 2.1 Use Cases

In this research work, we automate the prediction and the tuning of options of the Linux kernel to optimise its binary size. There are numerous use-cases for tuning options related to binary size of the kernel motivating this task [26, 53], in particular:

- the Linux community has introduced the command make tinyconfig to produce one of the smallest possible kernel. Though a user cannot use such a kernel to boot a Linux system, it can be used as a starting point for *e.g.,* embedded systems in efforts to reduce the kernel size – see also Section 5;
- the kernel should run on very small systems (IoT) or old machines with limited resources;
- Linux can be used as the primary bootloader. The size requirements on the first-stage bootloader are more stringent than for a traditional running operating system;
- size reduction can improve flash lifetime, spare RAM and maximize performances;
- in terms of security, the attack surface can be reduced by removing the optional parts that are not really needed;
- the kernel should boot faster and consume less energy: though there is no empirical evidence for how kernel size relates to other non-functional properties, practitioners tend to follow the hypothesis that the higher the size, the higher the energy consumption;
- a supercomputing program may want to run at high performance entirely within the L2 cache of the processor. If the combination of kernel and program is small enough, it can avoid accessing main memory entirely.

When configuring a kernel, binary size is usually neither the only concern nor the ultimate goal. The minimisation of the kernel size has no interest if the kernel is unable to boot on a specific device. Size is rather part of a suitable tradeoff between hardware constraints, functional requirements, and other non-functional concerns (*e.g.,* security). Our work is a first step in this direction.

### 2.2 Documentation of Linux Options

The presence of logical constraints and subtle interactions between options further complicates the size prediction.

Currently, Linux kernel builders set values to options through a configurator [76] and store them into a *.config* file. But because of cross-cutting constraints, all combinations of options will not result in a successful build. It is up to the end-user to manually change the values of options based on the documentation of the kernel. The same applies when changing the options to modify the resulting binary size of compilation : the user is left alone with the documentation. In the kernel, this documentation consists in a set of Kconfig files – one per configuration option.

```
1  config LOCK_STAT
2    bool "Lock usage statistics"
3    depends on STACKTRACE_SUPPORT && LOCKDEP_SUPPORT ...
4    select LOCKDEP
5    select DEBUG_SPINLOCK ...
6    default n
7    help
8      This feature enables tracking lock contention points. For
9      more details, see Documentation/locking/lockstat.txt
10     This also enables lock events required by "perf lock",
11     subcommand of perf. If you want to use "perf lock", you also
12     need to turn on CONFIG_EVENT_TRACING. CONFIG_LOCK_STAT
13     defines  "contended" and "acquired" lock events.
14     (CONFIG_LOCKDEP defines "acquire" and "release" events.)
```

**Figure 1: Option LOCK_STAT (excerpt)**

For example, Figure 1 depicts the *KConfig* file describing the *LOCK_STAT* option. Out of this file, we know that: this boolean option (line 2) has several direct dependencies, like the option *LOCKDEP_SUPPORT* (line 3); when selected, this option activates several other options such as *LOCKDEP* (lines 4-5); by default, this option is not selected (line 6). But the documentation of this option (lines 7→14) does not give any indication about the impact of *LOCK_STAT* on the kernel size. Our work aims at completing this *Kconfig* file by quantifying its effect on the binary size.

### 2.3 Problem Summary

Use cases, Kconfig documentation, options values for default configurations, as well as past and ongoing initiatives provide evidence that options related to kernel size are an important issue for the Linux community. However, the human effort to document configuration options and maintain the knowledge over time and kernel evolutions is highly challenging. It is actually a well-known phenomenon reported for many software systems [62] that is due to the exponential number of possible configurations. To improve the situation, our objective is to predict the effects of options w.r.t. size so users can then make informed and guided configuration decisions. Users in charge of configuring the kernel should indeed have the maximum of flexibility to meet their specific requirements - directly related to size or not.

A more automated and scalable approach could thus be helpful to capture the essence of size-related options in the very large configuration space of Linux. Because building all configurations is infeasible, our idea is to learn from a sample of measured configurations. The approach is not novel and numerous configurable systems have considered this idea in the literature [57] but not with the scale of the Linux Kernel. We are unaware of approaches and studies that thoroughly apply learning techniques to interpret huge configuration spaces with thousands of options.

The challenge is to apply the right learning algorithm with the right balance between *accuracy* (our prediction is close enough to reality), *training time*, and *interpretability*, because we want to communicate to developers and users which options matter for the kernel size. By selecting influential features in large configuration spaces, the promise of the solution is to improve accuracy, training time, and interpretability.

## 3 RESEARCH QUESTIONS

The main research question is *RQ: How to predict the effect of Linux kernel options on its size?*, which can be decomposed in five main research questions:

- **(RQ1, state-of-the-art) How do state-of-the-art techniques perform on large configurable systems?** Prior to the study, we use various state-of-the-art algorithms on our dataset to assess how they can be used.
- **(RQ2, accuracy) How accurate is the prediction model with and without feature selection?** Depending on the training size and the option chosen in the dataset fed to the learning algorithm, the resulting model may produce more or fewer errors when predicting the size of unseen configurations. The greater the accuracy of the model, the more accurate the interpretation.
- **(RQ3, stability) How stable are important options?** If the options of the kernel that matter to predict its binary size are always changing according to our models, it might be harder to interpret the general effects of options. We want to ensure the stability of the list of important options.
- **(RQ4, training time) How much computational resources is saved with feature selection?** By selecting a subset of relevant options, learning algorithms operate on a smaller number of features at training time. Depending on the training size and the option chosen, the resulting model may need more or fewer computational resources to get trained. In this question, we estimate the cost of the interpretation.
- **(RQ5, interpretability) How do feature ranking lists, as computed by tree-based feature selection, relate to Linux knowledge?** We aim to compare the retrieved knowledge extracted from models with knowledge from the Linux community. Linux knowledge notably comes from the Kconfig documentation and experts' insights.

## 4 LEARNING WITH TREE-BASED FEATURE SELECTION

This section describes our experimental protocol. The whole process is depicted in Figure 2 and each step is detailed in the remainder of this section. Section 4.1 and Section 4.2 describe how to build a dataset of configuration measurements - upper part of Figure 2. Section 4.3 and Section 4.4 details our implementation of statistical learning and feature selection - middle part of Figure 2. Section 4.5 lists the metrics and methods used to interpret the effects of options on the Linux kernel sizes - bottom part of Figure 2.

### 4.1 Gathering configurations' data

*4.1.1 TuxML.* We have developed the tool TuxML to build the Linux kernel in the large *i.e.*, whatever options are combined.

TuxML uses Docker to host the numerous packages and tools needed to automatically compile, build the Linux kernel and measure its size. The upper part of Figure 2 presents an overview of the TuxML measurement process. The two main steps followed by TuxML to measure kernel binary sizes are as follows: (1) *Sampling configurations.* For this step, we relied on randconfig to randomly generate Linux kernel configurations. randconfig has the merit of generating valid configurations respecting the numerous constraints between options. It is also a mature tool that the Linux

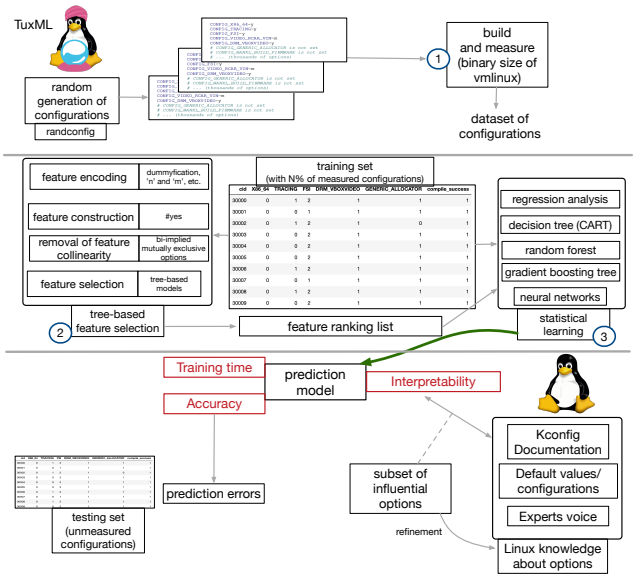


**Figure 2: From measuring to learning with tree-based feature selection**

community maintains and uses [46]. We use this sample to create a .config file. (2) *Kernel size measurement.* Given a set of .config files, TuxML builds the corresponding kernels. Throughout the process, TuxML measures the size of vmlinux, a statically linked executable file containing the kernel. We saved the configurations and the resulting sizes in a database. The outcome is a dataset of Linux configurations together with their binary size. Each line of the dataset is composed of a set of configuration option values and a quantitative value.

*4.1.2 Implementation.* We only focus on the kernel version 4.13.3 (release date: 20 Sep 2017). It is a stable version of Linux (*i.e.*, not a release candidate). Furthermore, we specifically target the x86-64 architecture *i.e.*, technically, all configurations have values `CONFIG_X86=y` and `CONFIG_X86_64=y`. During several months, we used a computing grid [1] to build and measure 95 854 random configurations. In total, we invested **15K hours of computation time**. A build took more than 9 minutes per configuration on average (standard deviation: 11 min). We removed configurations that do not build *e.g.*, due to configuration bugs of Linux.

### 4.2 Dataset

*4.2.1 Data description.* The number of possible options for x86 architecture is 12 797 options, but the 64 bits further restricts the possible values because some options are always set to *'y'* or *'n'* values. We observe that more than 3K options have a unique value mainly due to the presetting of `CONFIG_X86_64=y`. We use this opportunity to remove the options that have a unique value; it can be seen as a straightforward and effective way of doing feature selection. In total, 9286 options have more than one value *i.e.*, there are nine thousand predictors that can potentially have an effect on binary size. Each configuration is composed of 9,286 options together with one measurement (binary size of vmlinux). Finally, after adding the number of activated options as a feature, and the

[1] Called IGRIDA, see http://igrida.gforge.inria.fr/

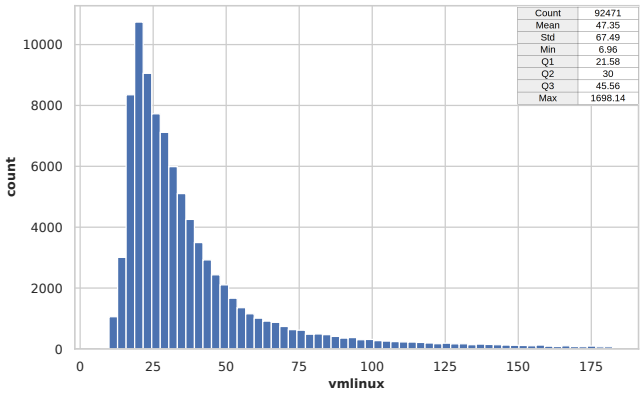


**Figure 3: Distribution of Linux kernel sizes (Mb, no outliers)**

elimination of identical feature, the dimension of the dataset (*i.e.,* number of features x number of configurations) is 8743 x 95 854.

*4.2.2 Performance distribution.* Before discussing the statistical learning, we briefly analyze the distribution of kernel sizes. The minimum size of vmlinux is 7Mb and corresponds to the size of tinyconfig, a predefined configuration that tries to preset option values to 'n' as much as possible. The maximum size is 1698Mb, the mean is 47Mb with a standard deviation of 67Mb. Figure 3 shows the distribution of vmlinux size (with 0.97 as quantile for avoiding extreme and infrequent size values). Most values are around 25Mb, showing an important variability in sizes.

*4.2.3 Feature encoding.* A first way to reduce the number of options is to properly encode options as features [14, 19] of the machine learning algorithm. The three possible values of options - 'y' for yes, 'n' for no and 'm' for module - can be encoded into numerical values; 'n' as 0, 'y' as 1 and 'm' as 2 is a first possible solution. However, some learning algorithms like linear regression will assume that two nearby values are more similar than two distant values (here 'y' and 'm' would be more similar than 'm' and 'n'). This encoding will also be confusing when interpreting the negative or positive weights of a feature. We observe that the 'm' value has no direct effect on the size since kernel modules are not compiled into the kernel and can be loaded as needed. Therefore, we consider that 'm' values have the same effect as 'n' values. As a result, the problem is simplified: an option can only take two values *i.e., "yes"* or *"no"*, 1 or 0. We also discarded 319 non-tri-state options.

*4.2.4 Feature construction.* We also construct aggregated feature to include in the model. For instance, we consider "the number of 'y' options" values as a feature, denoted #yes. The rationale is that the more options are included, the larger is the kernel. However, this feature is informative and may have a strong predictive power: it made explicit the domain knowledge about the relationship between binary size and number of options.

*4.2.5 Elimination of feature collinearity.* In machine learning, feature collinearity happens when two (or more) features are completely correlated. Feature collinearity is in fact a well-known, general issue in machine learning, especially for model interpretability [8, 13, 18, 48, 54]. Feature collinearity also becomes an issue when quantifying how much a feature is important to predict the target values. In order to mitigate feature collinearity, we grouped collinear options into a single feature.

## 4.3 Statistical learning problem

Predicting the size of the Linux kernel now becomes a supervised regression problem, *i.e.,* we aim to predict a quantitative variable out of features, built from configuration options.

*4.3.1 Formalisation.* Let us formalise the problem. First, we denote $p$ the number of configuration options and define the configuration space $\mathbb{C} = \{0,1\}^p$. Out of this space of configurations $\mathbb{C}$, we gather a subset of $d$ configurations, denoted $\mathbb{C}_\mathbb{S} \subset \mathbb{C}^d$. We separate $\mathbb{C}_\mathbb{S}$ into a training set $\mathbb{C}_\mathbb{S}^{tr}$ and a test set $\mathbb{C}_\mathbb{S}^{te}$, so $\mathbb{C}_\mathbb{S} = \mathbb{C}_\mathbb{S}^{tr} \bigoplus \mathbb{C}_\mathbb{S}^{te}$. Then, we denote :

- $F(\mathbb{C}, \mathbb{R}^+)$ all the functions of $\mathbb{C}$ in $\mathbb{R}^+$,
- $f : C \rightarrow R^+ \in F(\mathbb{C}, \mathbb{R}^+)$ the function affecting to any configuration $c \in \mathbb{C}$ its performance $f(c) \in \mathbb{R}^+$,
- $f(\mathbb{S})$ the distribution of performance of a set of configuration $\mathbb{S}$, verifying : $\forall\, \mathbb{S} \subset \mathbb{C}, f(\mathbb{S}) = \{f(c), c \in \mathbb{S}\}$.

With respect to these notations, the goal is to train a learning algorithm $\hat{f}$ estimating the function $f$ for each measured configuration of the training set $c \in \mathbb{C}_\mathbb{S}^{tr}$, *i.e.,* $\hat{f} = \underset{h \in F(\mathbb{C},\mathbb{R}^+)}{\operatorname{argmin}} L(f(\mathbb{C}_\mathbb{S}^{tr}), h(\mathbb{C}_\mathbb{S}^{tr}))$, where $L$ is a loss function.

The training set $\mathbb{C}_\mathbb{S}^{tr}$ is used to obtain a prediction model, while the testing set $\mathbb{C}_\mathbb{S}^{te}$ only tests the prediction performances of $\hat{f}$. The principle is to confront predicted values $\hat{f}(\mathbb{C}_\mathbb{S}^{te})$ to observed (*i.e.,* ground truth) values $f(\mathbb{C}_\mathbb{S}^{te})$.

*4.3.2 Learning algorithms.* We considered *Decision Tree*, *Random Forest*, *Gradient Boosting tree* [8], *Neural Networks* [20], *Performance-Influence Model* [22] and variants of *Regression*. We chose to work with this set of algorithms for two reasons. First, they have already been successfully used in the literature of configurable systems [57]. Linear regressions, decision trees, and random forests are the most used learning techniques in the literature. Second, we aim to gather strong baselines and explore the tradeoffs w.r.t. accuracy and interpretability. To do so, we considered *Lasso* [71], *Ridge*, and *Elastic Net* that perform embedded feature selection and/or regularization in order to enhance the prediction accuracy and interpretability of standard linear regressions. To investigate the effects of handling k-wise interactions as part of the learning process, we also included in our experiments: (1) *Polynomial* Regression, a linear regression that operates over polynomial combinations of the features with degree two, and (2) *Performance-Influence Model* [22][2].

*4.3.3 Hyperparameters.* Most of the selected algorithms are sensitive to the choice of *hyperparameters*, which may affect accuracy results. Otherwise, the best algorithm could be sub-optimal after the hyperparameter optimization. We optimize their hyperparameters, and explore a wide range of values as part of our study using grid search and cross-validation. The final hyperparameters and the different neural network topologies can be consulted online [69].

*4.3.4 Training and test.* For each learning method, we split our dataset of configurations and their associated size measures into multiple training and testing set of various sizes $N$ (10%, 50%, 80%, 90% of the dataset). For instance, when $N$ = 10% of the training set, we use 90% of the remaining configurations to test the predictions.

[2]Implemented with SPLConqueror, see https://github.com/se-passau/SPLConqueror, git commit 89b68ce

In all cases, we reproduced the experiment 5 times to mitigate the randomness and report on the average as well as standard deviation.

*4.3.5 Loss function.* We used the Mean Squared Error (MSE) as the loss function L for all these algorithms. For instance, the loss function evaluated on the training set $MSE_{tr}$ would be defined as follows w.r.t. the notations of Section 4.3.1:

$$MSE_{tr} = \frac{1}{\#(\mathbb{C}_{\mathbb{S}}^{tr})} \sum_{c \in \mathbb{C}_{\mathbb{S}}^{tr}} (f(c) - \hat{f}(c))^2$$

*4.3.6 Experimental infrastructure. Hardware* When measuring prediction errors (RQ2) or computation time (RQ4), we used one machine with an Intel Xeon E5-1630v3 4c/8t, 3, 7GHz, with 64GB DDR4 ECC 2133 MHz memory. *Reproducible environment.* We created an infrastructure on top of a Docker image [69]. The infrastructure takes a file as input that includes the specification of experiments and serializes the results for further analysis. *Machine learning libraries.* We rely on Python modules scikit-learn [9] and Tensorflow [2] to benefit from the state-of-the-art implementations. We also build an infrastructure around these libraries to automatically handle the feature engineering part, the control of hyperparameters, training set size, *etc. Bounds for training.* We needed to set up bounds to what is deemed reasonable for the experimentation. The first is about memory consumption, as we performed all our experiments on the same machine with 64GB of memory, which is already higher than the current personal computer standard - between 8 and 32GB. For CPU consumption, we put the limit at 10 days of computing. More details can be found online [69].

## 4.4 Tree-based feature selection

*4.4.1 Intuition.* A major problem when predicting the non-functional property of software variants is to identify the influence of options together with their interactions. Trying to naively list all interactions would only lead to a combinatorial explosion, especially at the scale of Linux with *e.g.,* more than $10^8$ possible interactions only for the degree 2. The approach taken by embedded feature selection methods such as decision tree, the tree-based feature selection, is good in that sense: the goal is to reduce the number of options $p$ by only taking the relevant ones in a step-wise manner. However, this reduction still operates over a very large number of variables at training time, which is sub-optimal in the case of Linux. In contrast, and to scale to the Linux case, we propose to identify a subset of important options **before applying machine learning algorithms or selecting the interactions in a step-wise manner**. The idea of this pre-process is that machine learning algorithms can then operate over a reduced number of variables at training time, thus reducing the overall cost of learning.

*4.4.2 Feature importance.* Some options of the Linux kernel might have little effects and can be removed without incurring much loss of information, because their impact on the resulting binary size is negligible. Knowing which configuration options are most predictive of size allows to focus on the interpretation of (interactions of) options that matter when predicting size. With that in mind, *feature importance* is a useful concept: it is the increase in the prediction error of the model after we permuted the feature's values [48]. For a decision tree, the importance of a feature is computed as the (normalized) total reduction of the splitting criterion (*e.g.,* Gini or entropy) brought by that feature. For random forest, *feature permutation importance* is computed through the observation of the effect on machine learning model accuracy of randomly shuffling each predictor variable [8, 48, 54].

*4.4.3 Feature Ranking List.* Once the feature importance of random forests is computed, the feature ranking list can be created. The feature ranking list orders features by importance. We use a threshold to select a subset of relevant options (*e.g.,* the top 500 of the list, by decreasing importance). This threshold allows one to control the reduced number of variables $p' \ll p$ to consider at training time. A learning algorithm can use the subset of features to eventually train a predictive model.

*4.4.4 Implementation.* We take the average ranking across 20 random forests to get the final feature ranking list. We needed to rely on the results of multiple random forests due to the instability of the model. Even with the same dataset and hyperparameters, the feature importances of two random forests were fairly different. For example, if we take the 300 most important features of the first random forest, we could only retrieve half of them in the second random forest top 300. An important factor that can influence the accuracy of a learning method with tree-based feature selection is $k$, the number of selected features within the feature ranking list. We make $k$ vary from 50 (*i.e.,* the 50 top influential features) to 8743 (the size of the entire list), in steps of 50. The goal is to identify how many features are ideally needed at training time for reaching high accuracy. We apply all learning methods of Section 4.3.2 with and without tree-based feature selection, and report on the difference of accuracy in Section 5.2.

*4.4.5 Feature Ranking List scenarios.* In the evaluation, we consider two usage scenarios. In RQ1 → RQ2, we consider both feature ranking list and actual training phases on the same portion of the dataset. In order to investigate the effect of the Feature Ranking List trained on a lower number of examples, we take a percentage of the dataset (*e.g.,* $N = 10\%$), create the Feature Ranking List and train a model using that List – all on the same training set. We repeat the process 5 times and report the mean MAPE. Specifically, we report prediction errors on Random Forest and Gradient Boosting Tree – two learning methods that proved to be highly accurate (see results of RQ1) – and on varying training size. In RQ3 → RQ5, the creation of a feature ranking list is done over a large portion of the dataset. This would represent a case when organizations (*e.g.,* KernelCI[3] for the Linux kernel) with computational resources share a Feature Ranking List to the community. In our experimental settings, we leverage a Feature Ranking List trained on 90% of the dataset with a set of random forests. The goal is to understand the potential of the feature ranking list in terms of computation time (RQ3), and interpretability (RQ4 and RQ5).

## 4.5 Metrics and measurements

*4.5.1 (RQ1-2) Accuracy.* To assess the accuracy of each methods, we rely on the mean absolute percentage error (MAPE), an interpretable and comparable metric. We choose the MAPE to depict our results because : it has the merit of being easy to understand and compare (it is a percentage); it handles the wide distribution and outliers of vmlinux size (Figure 3); it is commonly used in approaches about learning and configurable systems [57]. All reported accuracy values are made with this metric in order to have an easier

[3] See https://kernelci.org/

comparison. The MAPE of the testing set $MAPE_{te}$ would be defined as follows w.r.t. the notations of Section 4.3.1:

$$MAPE_{te} = \frac{100}{\#(\mathbb{C}_{\mathbb{S}}^{te})} \sum_{c \in \mathbb{C}_{\mathbb{S}}^{te}} \frac{\left| f(c) - \hat{f}(c) \right|}{f(c)} \%$$

*4.5.2 (RQ3) Stability.* The Feature Ranking List is created using the feature importance given by a model. The Random Forest relies on random selection and this can cause inconsistency in the feature importance. To measure the stability of Feature Ranking Lists, we count the number of common features in the top N features between two lists. The two lists have to be created under the same conditions (training set and hyperparameters) to be sure that only the randomness introduced by the machine learning algorithm causes the instability.

*4.5.3 (RQ4) Training time.* During experiments, we also collected the time needed to train each model. All reported times are given in seconds and have been measured on the same machine.

*4.5.4 (RQ5) Interpretability.* We quantitatively and qualitatively cross-checked the feature ranking list with Linux knowledge coming from the Kconfig documentation, expert insights (two developers of the Linux kernel), and tinyconfig. We report on options considered as influential by both sides, but also options absent in the list or in the Linux documentation.

## 5 RESULTS

We now present our results and answer the different research questions defined in Section 3 following the protocol of Section 4.

### 5.1 (RQ1) State-of-the-art performance

Most of the studied techniques could perform their task in the time and memory limits we had set. We however failed to get results from two of the techniques we have tried: Performance-Influence Model (implemented with SPLConqueror) and Polynomial regression. We speculate that the number of interactions to include in the model is too important for these two techniques, either in training time or in memory. Importing the Linux dataset into SPLConqueror raises an error about insufficient memory and cannot perform anything on the dataset[4]. Similarly, polynomial regression integrates interactions among features and does not scale for a degree 2 due to insufficient memory. It confirms the theoretical observations of Section 4.4. Therefore, we propose to apply feature selection before computing the feature interactions.

**(RQ1, state-of-the-art) How do state-of-the-art techniques perform on large configurable systems?** Performance-Influence model and Polynomial Regression cannot handle the Linux dataset in reasonable time and memory limits. They should be prohibited or adapted when predicting the performance of huge configurable systems.

[4]We used SPLConqueror from commit 9b68ce on Ubuntu 20.04 LTS and got the message *"System.OutOfMemoryException: Insufficient memory to continue the execution of the program."*

| Algorithm | Without tree-based feature selection | | | |
|---|---|---|---|---|
| | N=10 | N=50 | N=80 | N=90 |
| OLS Regression | 74.54±2.3 | 61.9 ± 1.14 | 50.37±0.57 | 49.42±0.08 |
| Lasso | 34.13±1.38 | 36.58±1.04 | 38.07±0.08 | 38.04±0.17 |
| Ridge | 139.63±1.13 | 62.42±0.08 | 55.75±0.2 | 51.78±0.14 |
| ElasticNet | 79.26±0.9 | 80.58±0.77 | 80.57±0.71 | 80.34±0.53 |
| Decision Tree | 15.18 ± 0.13 | 11.32±0.07 | 10.61± 0.10 | 10.48± 0.15 |
| Random Forest | 12.5±0.19 | 9.27±0.07 | 8.6±0.07 | 8.4 ±0.07 |
| GB Tree | 11.13±0.23 | 7.70±0.04 | 7.02±0.05 | 6.83±0.10 |
| N. Networks | - | 9.46 ± 0.15 | 8.29 ± 0.18 | 8.08 ± 0.12 |
| Polynomial | - | - | - | - |
| Algorithm | With tree-based feature selection | | | |
| | N=10 | N=50 | N=80 | N=90 |
| OLS Regression | 43.56±1.48 | 40.23±0.22 | 39.56±0.39 | 39.29±0.48 |
| Lasso | 35.18±0.45 | 39.28±1.06 | 38.28±0.04 | 38.61±0.81 |
| Ridge | 43.52±1.41 | 40.2±0.27 | 39.53±0.33 | 39.24±0.43 |
| ElasticNet | 79.66±2.11 | 81.0±0.24 | 80.84±0.6 | 81.45±0.2 |
| Decision Tree | 13.97±0.08 | 10.75±0.05 | 10.07±0.09 | 9.91±0.12 |
| Random Forest | 10.44±0.12 | 8.2±0.02 | 7.67±0.06 | 7.54±0.02 |
| GB Tree | 8.34±0.07 | 5.89±0.02 | 5.50±0.02 | 5.41±0.02 |
| N. Networks | - | 6.17 ± 0.09 | 5.77 ± 0.04 | 5.71 ± 0.05 |
| Polynomial | 24,65 ± 1.23 | 20.49 ± 0.24 | 21.53 ± 0.1 | 20.86 ± 0.04 |

**Table 1: MAPE of learning algorithms for the prediction of the Kernel size, without and with feature selection, with N the percentage of the dataset used as training**

### 5.2 (RQ2) Accuracy

*5.2.1 What is the best algorithm to learn the Linux kernel's size?* In Table 1, we report the MAPE (and its standard deviation) of multiple statistical learning algorithms, on various training set sizes (N), with and without tree-based feature selection. We observe that algorithms based on linear regression (Lasso, Ridge, Elasticnet) do not work well, having a MAPE of more than 30% whatever the training set size or the feature selection, showing that the size problem is not only an additive problem. Some do not even take advantage of a bigger training set, like Lasso which is even increasing its MAPE from 34% to 38% when the training set size goes from 10% to 90%. The results show that linear regressions are not suited for Linux and that the problem of predicting size cannot be trivially resolved with a simple additive model. Tree-based algorithms (Decision Tree, Random Forest, Gradient Boosting Tree) tend to work far better (MAPE lower than 15%) and effectively take advantage of more data to train on. As a base of comparison, we also report results from Neural Networks which are better than most of algorithms when fed with a big enough training set. However, 10% of the data in the training set was not enough to make a multiple layer network converge, thus explaining the *"-"* in the *"N = 10 column"*.

*5.2.2 How many configuration options do we need to learn the Linux kernel's size?* Figure 4 aims to show the influence on the accuracy of (1) the number $k$ of selected features and (2) the training set percentage (N) of the full dataset. In particular, it depicts how Random Forest performs when varying $k$ and N. We observe that for a training set size of N=10% (9, 500 configurations), the accuracy peaks (*i.e.,* lowest errors rate) when $k$=100 with a MAPE of 9.57, and consistently increases with more features. For a training set size of N=50% (47, 500), accuracy peaks when $k$=200 with a MAPE of 7.74. For N=90% (85, 500), we reach a MAPE of 7.29 with $k$=250. In Figure 4 and independently from the training set size, we consistently observe the lowest MAPE when $k$ is in the range 100 → 250.

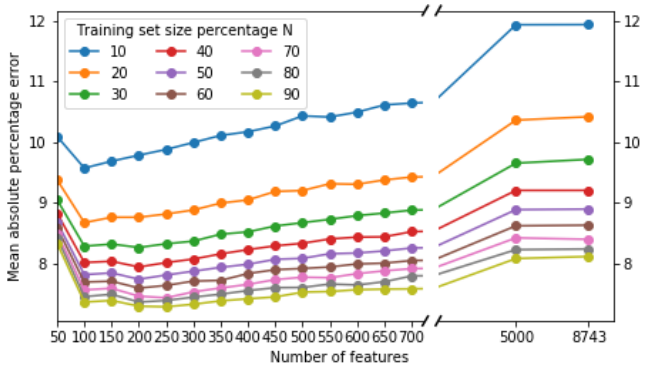


**Figure 4: Evolution of MAPE w.r.t. the number $k$ of selected features and the training set percentage $N$ (Random Forest).**

This optimal number of selected features depends on the algorithm used. Table 1 reports the accuracy results with feature selection of the optimal number for each learning method. For example, for Ridge or ElasticNet, it is in the range 250 → 300, 350 → 400 for Lasso, 1400 → 1600 for Gradient boosting tree and 1500 for Neural Networks. Given these results, we can say that out of the thousands of options of Linux kernel, only a few hundreds actually influence its binary size. From the machine learning point of view, the other features do not bring any more information and even make the model worse by biasing it. The effect of the constructed feature #yes is also quite noticeable, by improving accuracy on tree-based solution by at least 2 points at every training set size. Overall, the approach of reducing the dimensionality of configuration data is validated: feature engineering and selection are beneficial.

| Algorithm | With tree-based Feature Selection | | | | |
|---|---|---|---|---|---|
| | N=10 | N=20 | N=50 | N=80 | N=90 |
| Random Forest | 11.70±0.02 | 9.73±0.02 | 7.97±0.01 | 7.53±0.01 | 7.49±0.01 |
| GB Tree | 8.64±0.10 | 7.20±0.07 | 6.16±0.29 | 5.21±0.12 | 5.13±0.04 |

**Table 2: MAPE of different learning algorithms for the prediction of vmlinux size, with a Feature Ranking List based on the same training set (N = % of the dataset used as training)**

*5.2.3 What is the effect of lower training size for Feature Ranking List?* Table 2 shows the MAPE of models trained on different training set sizes, with the help of a Feature Ranking List itself created with the same training set. The impact of having less measurements to create the Feature Ranking List can be noticeable but is non significant overall. The only noteworthy changes compared to Table 1 (bottom table, with tree-based feature selection) is that at 10% of training set size, the MAPE of Random Forest is 1.3 point higher (10.44% vs 11.70%). However, it is still better than without feature selection (12.5%). In this practical scenario, the use of tree-based feature selection gives better accuracy results.

**(RQ2, accuracy) How accurate is the prediction model with and without feature selection?** We find a sweet spot where only 200 → 300 features are sufficient to efficiently train a Random Forest and a Gradient Boosting Tree to obtain a prediction model that outperforms other baselines (6% prediction errors for 40K configurations). We observe similar feature selection benefits for any training set size and tree-based learning algorithms. Overall, the use of tree-based feature selection is to be recommended when training a model.

## 5.3 (RQ3) Stability

*5.3.1 First benefit of tree-based selection : a stable Feature Ranking List.* At first, we simply created a list using only one Random Forest. However, when comparing two lists created using two different models yet trained in the same conditions, we could observe major differences already in the top 10 features, with only 9 features being in that top 10 in both. When comparing the top 300, we could only observe from 100 (with feature collinearity) up to 130 (after elimination of feature collinearity) common features. This instability would be a problem so we changed the way we create the Feature Ranking List by using an ensemble of Random Forests. By using the average feature importance of 20 Random Forests, the resulting Feature Ranking List shows a stability of 10 common features in the top 10 and 286 common features in the top 300. Despite not being perfectly stable, relying on an ensemble of Random Forests proves itself far more stable then using a single model. When comparing Feature Ranking Lists with and without eliminating feature collinearity, we can observe its influence. For instance, the group #153 is in the $30^{th}$ position, so a quite important feature, but when each of the options composing the group are taken individually, namely `IOSCHED_NOOP`, `SBITMAP`, and `BLOCK`, they are positioned $136^{th}$, $109^{th}$, and $130^{th}$, which shows a significant decrease in their computed importance.

**(RQ3, stability) How stable are important options?** Using an ensemble of Random Forest allows the creation of a far more stable list, with more than 95% common features in top 300 between multiple lists.

| Algorithm | Features | N=10% | N=20% | N=50% | N=80% | N=90% |
|---|---|---|---|---|---|---|
| Random Forest | 100 | 13 | 14 | 26 | 43 | 49 |
| | 200 | 14 | 21 | 50 | 86 | 97 |
| | 300 | 17 | 29 | 75 | 129 | 147 |
| | 400 | 21 | 37 | 101 | 173 | 199 |
| | 500 | 25 | 47 | 126 | 220 | 254 |
| | **8743[1]** | **383** | **814** | **2342** | **4030** | **4632** |
| GB Tree | 250 | 51 | 108 | 317 | 555 | 604 |
| | 500 | 100 | 210 | 630 | 1089 | 1252 |
| | 750 | 148 | 328 | 993 | 1705 | 1952 |
| | 1000 | 202 | 449 | 1360 | 2341 | 2679 |
| | 1250 | 266 | 580 | 1759 | 3029 | 3562 |
| | 1500 | 328 | 719 | 2169 | 3755 | 4350 |
| | **8743[1]** | **1693** | **3675** | **11234** | **19302** | **22090** |

**Table 3: Computing time (in seconds) with and without Feature Selection. N is the training set percentage.**

## 5.4 (RQ4) Training time

Our obtained results show that tree-based feature selection is promising w.r.t. accuracy and interpretability, but also beneficial on computational resources.

*5.4.1 Second benefit of tree-based feature selection : a shorter training time.* During the experiments, we observed that training the model without feature selection took a lot of time to compute (up to 24 hours for some settings of boosting trees). To further assess the effect of feature selection, we performed a controlled experiment. We measured the computation time for Random Forest and Gradient Boosting Tree, on multiple number of features and training set sizes, and reported the results in Table 3. For Random Forest, we report a 4, 632 seconds (77 minutes) computation over 85K rows of

the dataset (N=90%), with all features. With 200 features selected, we report a computation time of 97 seconds, 147 seconds for 300 features and 254 seconds for 500 features. The time reduction was respectively 48, 31, 18 fold less than without feature selection. With Gradient Boosting Tree, on all features and 85k rows of the dataset, we report a computation time of 22, 090 seconds (more than 6 hours). In the same conditions, for different number of selected features, we observed a difference in time. With 500 features selected, we report a computation time of 1252 seconds (21 minutes), 2679 seconds (45 minutes) for 1000 features, and 4350 seconds (72 minutes) for 1500 features. The time reduction was respectively 17, 8, 5 fold less than without feature selection. The computation time difference between Random Forest and Gradient Boosting Tree is explained by the capability of Random Forest to use multiple threads while Gradient Boosting Tree is not multi-threaded. In the context of intensive search of hyperparameters, it is perfectly possible to run multiple Gradient Boosting Trees in parallel, effectively using multiple threads. For reference, the end-to-end process takes 13 hours per fold, from finding good hyperparameters for Random Forests to create the Feature Ranking List, to hyperparameters optimisation of Gradient Boosting Tree and training the final model. These results show big savings in computation time, especially for Random Forest. We are convinced that similar benefits of tree-based feature selection can be obtained for other learning algorithms.

**(RQ4, training time) How much computational resources is saved with feature selection?** We find that tree-based feature selection speeds the model training at least 5 times up to 48 times. This way, we show that it is possible to greatly reduce the time needed to produce a model without sacrificing accuracy.

## 5.5 (RQ5) Interpretability

Contrary to Neural networks, Gradient Boosting Tree or Random Forest are algorithms with built-in interpretability. There, we can extract a list of features that can be ordered by their importance with respect to the property of interest (here size). So the question is: How do feature ranking lists, as computed by tree-based feature selection, relate to Linux knowledge?

*5.5.1 Conformance with documentation.* We cross-check the feature ranking list to the 147 options referring to size in the Kconfig documentation (see Section 2.2). First, we notice that 31 options have a unique value in our dataset: randconfig was unable to diversify the values of some options and therefore the learning phase cannot infer anything. We see it as an opportunity to further identifying influential options. As a proof of concept, we sample thousands of new configurations with and without option `KASAN_INLINE` activated (in our original dataset, `KASAN_INLINE` was always set to 'n'). We did observe size increase (20% on average), suggesting that the documentation could help identifying influential options we have missed due to randconfig. However, the documentation can put us on the wrong track: we have also tried for 5 other options and did not observe a significant effect on size [69]. Among the resulting 116 options (147 − 31), we found that 7 are in the top 50, 6 are in the range 50 → 250, 6 are in the rang 250 → 500, and 28 in the range 500 → 1500. 60% of options are also beyond the rank 1500, hence not considered after feature selection. We identified two possible explanations. First, the effect on size is simply negligible: It is explicitly stated as such in the documentation (*"This will increase the size of the kernelcapi module by* 20 *KB"* or *"Disabling this option saves about* 300 *bytes"*). Second, some option values are not distributed uniformly (*e.g.,* 98% *'y'* and 2% *'n'* value) in our dataset: the learning phase needs more diverse instances.

*5.5.2 Finding of undocumented options.* A consequence from the confrontation with Kconfig is that the *vast majority of influential options is either not documented or not referring to size.* In order to further understand this trend, we analyze the top 50 options in the feature ranking list (representing 95% of the feature importance in the Random Forest): only 7 options are documented as having a clear influence on size; our investigations and exchanges with domain experts[5] show that the 43 remaining options are either (1) necessary to activate other options; (2) the underlying memory used would be based on the size of the driver; (3) chip-selection configuration (you cannot run the kernel on the indicated system without activating this option); (4) related to compiler coverage instrumentation, which will affect lots (possible all) code paths; (5) debugging features; (6) related to DRM (for Direct Rendering Manager) and GPU drivers.

*5.5.3 Revisiting tinyconfig.* In our dataset, we observe that tinyconfig [53] is by far the smallest kernel (≈ 7Mb). The second smallest configuration is ≈ 11Mb. That is, despite 90K+ measurements with randconfig, we were unable to get closer to 7Mb. Can our prediction model explain this significant difference (≈ 4Mb)? A first hypothesis is that the four pre-set options of tinyconfig have an important impact on the size. We observe that our prediction model indeed ranks `CC_OPTIMIZE_FOR_SIZE`, one of the four pre-set options, in the top 200. The other three remaining options have no significant effects according to our model and cannot explain the ≈ 4Mb difference. In fact, there is a more simple explanation: the strategy of tinyconfig consists in minimizing #yes. According to our prediction model, #yes is highly influential feature. We notice that the number of 'y' options of tinyconfig is 224 while the second smallest configuration exhibits 646 options: the difference is significant. Furthermore, tinyconfig deactivates many top-influential options like `KASAN` or `DEBUG_INFO` that have a positive effect on the binary size when activated. We can conclude that the insights of the predictive model are consistent with the heuristic of tinyconfig.

*5.5.4 Collinearity and interpretability.* The top 50 of our feature ranking list exhibits 4 groups with collinear features. Thanks to removal of feature collinearity, we have automatically identified cases in which some options can be grouped together (*e.g.,* we can remove `KASAN_OUTLINE` and only keep `KASAN`). We found that feature collinearity is beneficial for two reasons (1) we do not miss influential features since the importance is not split among features; (2) experts only have to review a group of features instead of unrelated and supposedly independent features.

*5.5.5 Documentation-based feature selection.* Instead of using the top $X$ features from the Features Ranking List of a tree-based feature selection, the 147 options referring to size in the Kconfig documentation (see Section 2) can be used to form the feature list fed to the actual training. This scenario corresponds to feature selection

[5] The full data is available on the companion web site [69]

realized with expert knowledge. We use all the 147 options in the Linux documentation and only them (considering that an option is encoded as a feature).

The resulting prediction model exhibits a MAPE of 23.6% for Gradient Boosting Tree (the best learning algorithm) and over 90% of the dataset for training. It is more than 4 times the error rate of using the Feature Ranking List used with tree-based feature selection. This show that documentation alone is not suited for feature selection. It is in line with previous observations: the Linux documentation is incomplete (*i.e.,* some important features are missing) and describes features that have little influence on size.

**(RQ5, interpretability) How do feature ranking lists, as computed by tree-based feature selection, relate to Linux knowledge?** Thanks to our prediction model, we have effectively identified a list of important features that are consistent with the options and strategy of tinyconfig, and Linux knowledge (Kconfig documentation and expert insight). We also found options that can be used to refine or augment the incomplete documentation of the Linux kernel.

## 6 THREATS TO VALIDITY

**Internal Validity.** The selection of the learning algorithms and their parameter settings may affect the accuracy and influence interpretability. To reduce this threat, we selected the most widely used learning algorithms that have shown promising results in this field [57] and for a fair comparison we searched for their best parameters. We deliberately used random sampling over the training set for all experiments to increase internal validity. For each sample size, we repeat the prediction process 5 times. For each process, we split the dataset into training and testing which are independent of each other. To assess the accuracy of the algorithms and thus its interpretability, we used MAPE since most of the state-of-the-art works use this metric for evaluating the effectiveness of performance prediction algorithm [57]. Another threat to validity concerns the (lack of) uniformity of randconfig. Indeed randconfig does *not* provide a perfect uniform distribution over valid configurations [46]. The strategy of randconfig is to randomly enable or disable options according to the order in the Kconfig files. It is thus biased towards features higher up in the tree. The problem of uniform sampling is fundamentally a satisfiability problem. To the best of our knowledge, there is no robust and scalable solution capable of comprehensively translating Kconfig files of Linux into a format usable by a SAT solver. Second, uniform sampling either does not scale yet or is not uniform [10, 11, 16, 58]. So we had to stick with randconfig. We see randconfig as a *baseline* widely used by the Linux community [47, 59]. The computation of feature importance is subject to some debates and some implementation over random forest, including the scikit-learn's one we rely on, may be biased [8, 18, 48, 54]. This issue may impact our experiments *e.g.,* we may have missed important options. To mitigate this threat, we have computed feature importance with repetitions (*i.e.,* 20 times). Our observation is that the top influential options then remain very similar [69] (see RQ3). As future work, we plan to compare different techniques to compute feature importance [8, 18, 48, 54].

**External Validity.** A threat to external validity is related to the target kernel version and architecture (x86) of Linux. Because we rely on the kernel version 4.13.3 and the non-functional property binary size, the results may be subject to this specific version and quantitative property. However, a generalization of the results for other non-functional properties (*e.g.,* boot time) would require additional experiments. It is actually a challenge *per se* to scale up such experiments. Binary size has the merit of being independent of the hardware: we do not need to control the heterogeneity of machines or repeat the measurements. Binary size has also proved to be a non-trivial property to learn. Overall, we focused on a single version and property to be able to make robust and reliable statements about whether learning approaches can be used in such settings.

## 7 RELATED WORK

### 7.1 Research work

*7.1.1 Linux kernel and configurations.* Several empirical studies [3, 4, 7, 12, 38, 39, 46, 49, 50, 55, 81] have considered different aspects of Linux (build system, variability implementation, constraints, bugs, compilation warnings). Nadi *et al.* [50] mined the Linux repository in the quest of so-called variability anomalies (*e.g.,* mapping code to an invalid configuration). She *et al.* [41] reverse engineer the feature model of Linux. Nadi *et al.* [49] extract constraints among options using various techniques. Passos *et al.* [55] analyze the evolution of the configuration space together with the variability implementation of Linux. In [7], authors perform an empirical study of unspecified dependencies in make-based build systems (including Linux). Lawall *et al.* [38, 39] propose automated techniques to support contributors work in reviewing and testing patches. However, these studies did not target the binary size of the kernel or did not concretely build configurations in the large. Melo *et al.* compiled 21K valid random Linux kernel configurations for an in-development and a stable version (version 4.1.1) [46]. The goal of the study was to quantitatively analyze configuration-dependent *warnings*, not binary size. Prior works considered only a few options and configurations. Sincero *et al.* [68] considered 352 options and 146 random configurations for the non-functional property scheduling performance. Siegmund *et al.* [65] considered 25 options and 100 random configurations for binary size.

*7.1.2 Machine learning for configurable systems.* Siegmund *et al.* [63] introduced a learning method called performance-influence model. Feature-forward selection and multiple linear regression are used in a stepwise manner to shrink or keep terms representing options or interactions. Though performance-influence models have been used in various settings [30, 34, 64, 74] [29, 32, 33], it is yet to be proven whether the method is suited for the scale of Linux. Specifically, the number of possible interactions in Linux is huge. As stated in [34], for $p$ options, there are $p$ possible main influences, $p \times (p-1)/2$ possible pairwise interactions, and an exponential number of higher-order interactions among more than two options. In the worst case, all 2-wise or 3-wise interactions among the 9K+ options are included in the model, which is computationally intractable. Even if a subset of options is kept, there is a combinatorial explosion of possible interactions. It may hinder the scaling of the method or dramatically increase the training time. Kolesnikov *et*

*al.* [34] reported that it take hundreds of minutes for systems with less than 30 options, which is far from 9K+ options.

There have been recent attempts about learning the configuration space of the Linux Kernel [42, 43]. We consider the problem of predicting properties of any configuration, whereas Martin et al. [43] consider the problem of specializing the configuration space. The former is a regression problem, the latter a classification one. The metrics and the loss functions to optimize are thus different (balanced accuracy vs MAPE). The considered algorithms also differ: Martin et al. only considered decision trees, while this paper consider a wide range of learning algorithms. Another key difference of our work is that we study in detail the quality of feature selection: (1) how many features are needed to reach a good trade-off between accuracy, interpretability, and computation time; (2) how features considered as influential relate to domain knowledge. In [42], Martin et al. learn performance models over the binary size of different Linux versions (from 4.13 to 5.8). In contrast to our study, feature selection and learning over a reduced set of options are not considered. We could combine their work to ours to see how the effects of options evolve across versions.

*7.1.3 Quality assurance and configuration sampling.* There has been a large body of work that demonstrates the need for configuration-aware testing techniques and proposes methods to sample and prioritize the configuration space [24, 31, 40, 44, 45, 52, 61, 70, 79]. In our study and similarly as in [6, 46, 65, 66], we simply reuse randconfig and did not employ a sophisticated strategy to sample the configuration space. As previously discussed, an exciting research direction is to apply state-of-the-art techniques over Linux but several challenges are ahead.

*7.1.4 Interpretability, machine learning and configurable systems.* Interpretability of prediction models is an important research topic [48] in numerous domains, with many open questions related to their assessment or computational techniques. Only a few studies have been conducted in the context of configurable systems [15, 17, 30, 34, 67, 68, 73]. These studies aim at learning an accurate model that is fast to compute and simple to interpret. Few works [30, 73] use similarity metrics to investigate the relatedness of the source and target environments to transfer learning, while others [15, 17, 34, 67, 68] use size metrics as insights of interpretability for pure prediction. Different from these works, we investigated the relatedness of the Linux documentation, tiny kernel pre-defined configuration, and Linux community knowledge with the results reached for several state-of-the-art learning approaches. We find a sweet spot between accuracy and interpretability thanks to feature selection; we also confront the interpretable information with evidence coming from different sources (see RQ5).

*7.1.5 Code reduction.* Software debloating has been proposed to only keep the features that users utilize and are deem necessary with several applications and promising results (operating systems, libraries, Web servers Nginx, OpenSSH, *etc.*) [25, 36, 37]. Software debloating can be used to further reduce the size of the kernel at the source code level, typically when the configuration is fixed and set up. Our approach only operates at the configuration level and is complementary to code debloating. Beyond configurable systems, feature selection has attracted increasing attention in software engineering in general (*e.g.,* for defect prediction [21, 35, 51]).

## 7.2 Community attempts

We review (informal) initiatives in the Linux community that are dealing with kernel sizes. The Wiki[1] provides guidelines to configure the kernel and points out numerous important options related to size. However the page is no longer actively maintained since 2011. Tim Bird (Sony) presented "Advanced size optimization of the Linux kernel" in 2013. Josh Triplett (Intel) introduced tinyconfig at Linux Plumbers Conference 2014 ("Linux Kernel Tinification") and described motivating use-cases. The leitmotiv is to leave maximum configuration room for useful functionality while exploiting opportunities to make the kernel as small as possible. It led to the creation of the project http://tiny.wiki.kernel.org. The last modifications were made 5 years ago on Linux versions 3.X https://git.kernel.org/pub/scm/linux/kernel/git/josh/linux.git/. Pieter Smith (Philips) gave a talk about "Linux in a Lightbulb: How Far Are We on Tinification (2015)". Michael Opdenacker (Bootlin) described the state of Linux kernel size in 2018. According to these experts, techniques for size reduction are broad and related to link-time optimization, compilers, file systems, strippers, *etc.* In many cases, a key challenge is that configuration options spread over different files of the code base, possibly across subsystems [3, 4, 7, 46, 49, 55, 56].

# 8 CONCLUSION

Is it possible to learn the essence of a gigantic space of software configurations with respect to a given property of interest? We addressed an instance of this problem through the prediction of Linux kernel sizes for the x86_64 processor architecture, dealing with over 9,000 options and thus such a huge configuration space. We invested 15K hours of computation to build and measure 95 854 Linux kernel configurations and found that tree-based feature selection pays off: (1) the accuracy is better with than without tree-based feature selection (2) the training time is decreased (3) the identification of influential options is consistent with, and can even improve, the expert knowledge about Linux kernel configuration.

**Acknowledgments.** This research was partially funded by the ANR-17-CE25-0010-01 VaryVary project.

**Open Science.** We provide a replication bundle in https://github.com/tuxml/size-analysis, with explanations about how to use TuxML, how to access the dataset and how to execute our code.

# REFERENCES

[1] URL https://elinux.org/Kernel_Size_Tuning_Guide
[2] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015). URL https://www.tensorflow.org/. Software available from tensorflow.org
[3] Abal, I., Brabrand, C., Wasowski, A.: 42 variability bugs in the linux kernel: a qualitative analysis. In: ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014, pp. 421–432 (2014). DOI 10.1145/2642937.2642990. URL https://doi.org/10.1145/2642937.2642990
[4] Abal, I., Melo, J., Stănciulescu, x., Brabrand, C., Ribeiro, M., Wasowski, A.: Variability bugs in highly configurable systems: A qualitative analysis. ACM Trans. Softw. Eng. Methodol. **26**(3), 10:1–10:34 (2018). DOI 10.1145/3149119. URL http://doi.acm.org/10.1145/3149119
[5] Abbasi, E.K., Hubaux, A., Acher, M., Boucher, Q., Heymans, P.: The anatomy of a sales configurator: An empirical study of 111 cases. In: Advanced Information

Systems Engineering - 25th International Conference, CAiSE 2013, Valencia, Spain, June 17-21, 2013. Proceedings, pp. 162–177 (2013)
[6] Acher, M., Martin, H., Alves Pereira, J., Blouin, A., Eddine Khelladi, D., Jézéquel, J.M.: Learning From Thousands of Build Failures of Linux Kernel Configurations. Technical report, Inria ; IRISA (2019). URL https://hal.inria.fr/hal-02147012
[7] Bezemer, C., McIntosh, S., Adams, B., Germán, D.M., Hassan, A.E.: An empirical study of unspecified dependencies in make-based build systems. Empirical Software Engineering **22**(6), 3117–3148 (2017). DOI 10.1007/s10664-017-9510-8. URL https://doi.org/10.1007/s10664-017-9510-8
[8] Breiman, L.: Random forests. Machine learning **45**(1), 5–32 (2001)
[9] Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., VanderPlas, J., Joly, A., Holt, B., Varoquaux, G.: API design for machine learning software: experiences from the scikit-learn project. In: ECML PKDD Workshop: Languages for Data Mining and Machine Learning, pp. 108–122 (2013)
[10] Chakraborty, S., Fremont, D.J., Meel, K.S., Seshia, S.A., Vardi, M.Y.: On parallel scalable uniform SAT witness generation. In: Tools and Algorithms for the Construction and Analysis of Systems TACAS'15 2015, London, UK, April 11-18, 2015. Proceedings, pp. 304–319 (2015)
[11] Chakraborty, S., Meel, K.S., Vardi, M.Y.: A scalable and nearly uniform generator of sat witnesses. In: International Conference on Computer Aided Verification, pp. 608–623. Springer (2013)
[12] Dintzner, N., van Deursen, A., Pinzger, M.: Analysing the linux kernel feature model changes using fmdiff. Software and System Modeling **16**(1), 55–76 (2017). DOI 10.1007/s10270-015-0472-2. URL https://doi.org/10.1007/s10270-015-0472-2
[13] Dormann, C., Elith, J., Bacher, S., Carré, G., García Márquez, J., Gruber, B., Lafourcade, B., Leitao, P., Münkemüller, T., McClean, C., Osborne, P., Reneking, B., Schröder, B., Skidmore, A., Zurell, D., Lautenbach, S.: Collinearity: a review of methods to deal with it and a simulation study evaluating their performance. Ecography **36**(1), 27–46 (2013). DOI 10.1111/j.1600-0587.2012.07348.x
[14] Draper, N.R., Smith, H.: Applied regression analysis, vol. 326. John Wiley & Sons (1998)
[15] Duarte, F., Gil, R., Romano, P., Lopes, A., Rodrigues, L.: Learning non-deterministic impact models for adaptation. In: Proceedings of the 13th International Conference on Software Engineering for Adaptive and Self-Managing Systems, pp. 196–205. ACM (2018)
[16] Dutra, R., Laeufer, K., Bachrach, J., Sen, K.: Efficient sampling of SAT solutions for testing. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pp. 549–559 (2018). DOI 10.1145/3180155.3180248. URL http://doi.acm.org/10.1145/3180155.3180248
[17] Etxeberria, L., Trubiani, C., Cortellessa, V., Sagardui, G.: Performance-based selection of software and hardware features under parameter uncertainty. In: Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures, pp. 23–32. ACM (2014)
[18] Fisher, A., Rudin, C., Dominici, F.: All models are wrong, but many are useful: Learning a variable's importance by studying an entire class of prediction models simultaneously (2018)
[19] Friesel, D., Spinczyk, O.: Performance is not boolean: Supporting scalar configuration variables in nfp models. In: Tagungsband des FG-BS Frühjahrstreffens 2022. Gesellschaft für Informatik e.V., Bonn (2022). DOI 10.18420/fgbs2022f-03
[20] Géron, A.: Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems. " O'Reilly Media, Inc." (2017)
[21] Ghotra, B., McIntosh, S., Hassan, A.E.: A large-scale study of the impact of feature selection techniques on defect classification models. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 146–157. IEEE (2017)
[22] Grebhahn, A., Rodrigo, C., Siegmund, N., Gaspar, F.J., Apel, S.: Performance-influence models of multigrid methods: A case study on triangular grids. Concurrency and Computation: Practice and Experience **29**(17), e4057 (2017)
[23] Guo, J., Czarnecki, K., Apel, S., Siegmund, N., Wąsowski, A.: Variability-aware performance prediction: A statistical learning approach. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 301–311 (2013). DOI 10.1109/ASE.2013.6693089
[24] Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., Baudry, B.: Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack. Empirical Software Engineering (2018). DOI 10.1007/s10664-018-9635-4. URL https://doi.org/10.1007/s10664-018-9635-4
[25] Heo, K., Lee, W., Pashakhanloo, P., Naik, M.: Effective program debloating via reinforcement learning. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18, pp. 380–394. ACM, New York, NY, USA (2018). DOI 10.1145/3243734.3243838. URL http://doi.acm.org/10.1145/3243734.3243838
[26] https://tiny.wiki.kernel.org/: Linux kernel tinification. Last access: july 2019
[27] Hubaux, A., Heymans, P., Schobbens, P., Deridder, D., Abbasi, E.K.: Supporting multiple perspectives in feature-based configuration. Software and System Modeling **12**(3), 641–663 (2013). DOI 10.1007/s10270-011-0220-1. URL https://doi.org/10.1007/s10270-011-0220-1
[28] Hubaux, A., Xiong, Y., Czarnecki, K.: A user survey of configuration challenges in linux and ecos. In: Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings, pp. 149–155 (2012). DOI 10.1145/2110147.2110164. URL https://doi.org/10.1145/2110147.2110164
[29] Jamshidi, P., Cámara, J., Schmerl, B., Kästner, C., Garlan, D.: Machine learning meets quantitative planning: Enabling self-adaptation in autonomous robots. arXiv preprint arXiv:1903.03920 (2019)
[30] Jamshidi, P., Siegmund, N., Velez, M., Kästner, C., Patel, A., Agarwal, Y.: Transfer learning for performance modeling of configurable systems: An exploratory analysis. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, p. 497–508. IEEE Press (2017)
[31] Jin, D., Qu, X., Cohen, M.B., Robinson, B.: Configurations everywhere: Implications for testing and debugging in practice. In: Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014, pp. 215–224. ACM, New York, NY, USA (2014). DOI 10.1145/2591062.2591191. URL http://doi.acm.org/10.1145/2591062.2591191
[32] Kaltenecker, C., Grebhahn, A., Siegmund, N., Guo, J., Apel, S.: Distance-based sampling of software configuration spaces. In: Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE). ACM (2019)
[33] Kolesnikov, S., Siegmund, N., Kästner, C., Apel, S.: On the relation of external and internal feature interactions: A case study. arXiv preprint arXiv:1712.07440 (2017)
[34] Kolesnikov, S., Siegmund, N., Kästner, C., Grebhahn, A., Apel, S.: Tradeoffs in modeling performance of highly configurable software systems. Software & Systems Modeling **18**(3), 2265–2283 (2019). DOI 10.1007/s10270-018-0662-9. URL https://doi.org/10.1007/s10270-018-0662-9
[35] Kondo, M., Bezemer, C.P., Kamei, Y., Hassan, A.E., Mizuno, O.: The impact of feature reduction techniques on defect prediction models. Empirical Software Engineering pp. 1–39 (2019)
[36] Koo, H., Ghavamnia, S., Polychronakis, M.: Configuration-driven software debloating. In: Proceedings of the 12th European Workshop on Systems Security, EuroSec '19, pp. 9:1–9:6. ACM, New York, NY, USA (2019). DOI 10.1145/3301417.3312501. URL http://doi.acm.org/10.1145/3301417.3312501
[37] Kurmus, A., Sorniotti, A., Kapitza, R.: Attack surface reduction for commodity os kernels: Trimmed garden plants may attract less bugs. In: Proceedings of the Fourth European Workshop on System Security, EUROSEC '11, pp. 6:1–6:6. ACM, New York, NY, USA (2011). DOI 10.1145/1972551.1972557. URL http://doi.acm.org/10.1145/1972551.1972557
[38] Lawall, J., Muller, G.: Jmake: Dependable compilation for kernel janitors. In: 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017, pp. 357–366 (2017). DOI 10.1109/DSN.2017.62. URL https://doi.org/10.1109/DSN.2017.62
[39] Lawall, J., Muller, G.: Coccinelle: 10 years of automated evolution in the linux kernel. In: 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018., pp. 601–614 (2018). URL https://www.usenix.org/conference/atc18/presentation/lawall
[40] Lazreg, S., Cordy, M., Collet, P., Heymans, P., Mosser, S.: Multifaceted automated analyses for variability-intensive embedded systems. In: Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, pp. 854–865 (2019). URL https://dl.acm.org/citation.cfm?id=3339612
[41] Lotufo, R., She, S., Berger, T., Czarnecki, K., Wasowski, A.: Evolution of the linux kernel variability model. In: Proc. of SPLC'10, *LNCS*, vol. 6287, pp. 136–150 (2010)
[42] Martin, H., Acher, M., Lesoil, L., Jezequel, J.M., Khelladi, D.E., Pereira, J.A.: Transfer learning across variants and versions : The case of linux kernel size. IEEE Transactions on Software Engineering pp. 1–1 (2021). DOI 10.1109/TSE.2021.3116768
[43] Martin, H., Acher, M., Pereira, J.A., Jézéquel, J.M.: A Comparison of Performance Specialization Learning for Configurable Systems, p. 46–57. Association for Computing Machinery, New York, NY, USA (2021). URL https://doi.org/10.1145/3461001.3471155
[44] Medeiros, F., Kästner, C., Ribeiro, M., Gheyi, R., Apel, S.: A comparison of 10 sampling algorithms for configurable systems. In: Proceedings of the 38th International Conference on Software Engineering - ICSE '16, pp. 643–654. ACM Press, Austin, Texas, USA (2016)
[45] Meinicke, J., Wong, C., Kästner, C., Thüm, T., Saake, G.: On essential configuration complexity: measuring interactions in highly-configurable systems. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016, pp. 483–494 (2016). DOI 10.1145/2970276.2970322. URL https://doi.org/10.1145/2970276.2970322
[46] Melo, J., Flesborg, E., Brabrand, C., Wasowski, A.: A quantitative analysis of variability warnings in linux. In: Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '16, pp. 3–8. ACM (2016)
[47] Melo, J., Flesborg, E., Brabrand, C., Wąsowski, A.: A quantitative analysis of variability warnings in linux. In: Proceedings of the Tenth International Workshop

on Variability Modelling of Software-intensive Systems, pp. 3–8. ACM (2016)
[48] Molnar, C.: Interpretable Machine Learning. Lulu. com (2020)
[49] Nadi, S., Berger, T., Kästner, C., Czarnecki, K.: Where do configuration constraints stem from? an extraction approach and an empirical study. IEEE Trans. Software Eng. (2015)
[50] Nadi, S., Dietrich, C., Tartler, R., Holt, R.C., Lohmann, D.: Linux variability anomalies: What causes them and how do they get fixed? In: Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, pp. 111–120. IEEE Press, Piscataway, NJ, USA (2013). URL http://dl.acm.org/citation.cfm?id=2487085.2487112
[51] Nam, J., Pan, S.J., Kim, S.: Transfer defect learning. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 382–391. IEEE (2013)
[52] Niu, X., n. changhai, Leung, H.K.N., Lei, Y., Wang, X., Xu, J., Wang, Y.: An interleaving approach to combinatorial testing and failure-inducing interaction identification. IEEE Transactions on Software Engineering pp. 1–1 (2018). DOI 10.1109/TSE.2018.2865772
[53] Opdenacker, M.: Bof: Embedded linux size (2018). Embedded Linux Conference North-America
[54] Parr, T., Turgutlu, K., Csiszar, C., Howard, J.: Beware Default Random Forest Importances (2018). Last access: july 2019
[55] Passos, L., Queiroz, R., Mukelabai, M., Berger, T., Apel, S., Czarnecki, K., Padilla, J.: A study of feature scattering in the linux kernel. IEEE Transactions on Software Engineering (2018)
[56] Passos, L., Queiroz, R., Mukelabai, M., Berger, T., Apel, S., Czarnecki, K., Padilla, J.: A study of feature scattering in the linux kernel. IEEE Transactions on Software Engineering pp. 1–1 (2018). DOI 10.1109/TSE.2018.2884911
[57] Pereira, J.A., Martin, H., Acher, M., Jézéquel, J.M., Botterweck, G., Ventresque, A.: Learning software configuration spaces: A systematic literature review (2019)
[58] Plazar, Q., Acher, M., Perrouin, G., Devroey, X., Cordy, M.: Uniform sampling of sat solutions for configurable systems: Are we there yet? In: ICST 2019 - 12th International Conference on Software Testing, Verification, and Validation, pp. 1–12. Xian, China (2019). URL https://hal.inria.fr/hal-01991857
[59] Rothberg, V., Dietrich, C., Ziegler, A., Lohmann, D.: Towards scalable configuration testing in variable software. In: ACM SIGPLAN Notices, vol. 52, pp. 156–167. ACM (2016)
[60] Sarkar, A., Guo, J., Siegmund, N., Apel, S., Czarnecki, K.: Cost-efficient sampling for performance prediction of configurable systems (t). In: ASE'15 (2015)
[61] Sayagh, M., Kerzazi, N., Adams, B.: On cross-stack configuration errors. In: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, pp. 255–265 (2017). DOI 10.1109/ICSE.2017.31. URL https://doi.org/10.1109/ICSE.2017.31
[62] Sayagh, M., Kerzazi, N., Adams, B., Petrillo, F.: Software configuration engineering in practice: Interviews, survey, and systematic literature review. IEEE Transactions on Software Engineering (2018)
[63] Siegmund, N., Grebhahn, A., Apel, S., Kästner, C.: Performance-influence models for highly configurable systems. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 284–294 (2015)
[64] Siegmund, N., Grebhahn, A., Kästner, C., Apel, S.: Performance-influence models for highly configurable systems. In: ESEC/FSE'15 (2015)
[65] Siegmund, N., Rosenmüller, M., Kästner, C., Giarrusso, P.G., Apel, S., Kolesnikov, S.S.: Scalable prediction of non-functional properties in software product lines. In: Software Product Line Conference (SPLC), 2011 15th International, pp. 160–169 (2011)
[66] Siegmund, N., Rosenmüller, M., Kästner, C., Giarrusso, P.G., Apel, S., Kolesnikov, S.S.: Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. Inf. Softw. Technol. **55**(3), 491–507 (2013). DOI 10.1016/j.infsof.2012.07.020. URL https://doi.org/10.1016/j.infsof.2012.07.020
[67] Siegmund, N., Sobernig, S., Apel, S.: Attributed variability models: outside the comfort zone. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 268–278. ACM (2017)
[68] Sincero, J., Schroder-Preikschat, W., Spinczyk, O.: Approaching non-functional properties of software product lines: Learning from products. In: Software Engineering Conference (APSEC), 2010 17th Asia Pacific, pp. 147–155 (2010)
[69] Supplementary material (Web page): https://github.com/tuxml/size-analysis
[70] Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A classification and survey of analysis strategies for software product lines. ACM Computing Surveys (2014)
[71] Tibshirani, R.: Regression shrinkage and selection via the lasso. Journal of the Royal Statistical Society: Series B (Methodological) **58**(1), 267–288 (1996)
[72] Torvalds, L.: The linux edge. Communications of the ACM **42**(4), 38–38 (1999)
[73] Valov, P., Petkovich, J.C., Guo, J., Fischmeister, S., Czarnecki, K.: Transferring performance prediction models across different hardware platforms. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, pp. 39–50. ACM (2017)
[74] Weckesser, M., Kluge, R., Pfannemüller, M., Matthé, M., Schürr, A., Becker, C.: Optimal reconfiguration of dynamic software product lines based on performance-influence models. In: Proceeedings of the 22nd International Conference on Software Product Line, pp. 98–109. ACM (2018)
[75] Xiong, Y., Hubaux, A., She, S., Czarnecki, K.: Generating range fixes for software configuration. In: 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland, pp. 58–68 (2012). DOI 10.1109/ICSE.2012.6227206. URL https://doi.org/10.1109/ICSE.2012.6227206
[76] Xiong, Y., Hubaux, A., She, S., Czarnecki, K.: Generating range fixes for software configuration. In: 34th International Conference on Software Engineering (2012)
[77] Xiong, Y., Zhang, H., Hubaux, A., She, S., Wang, J., Czarnecki, K.: Range fixes: Interactive error resolution for software configuration. IEEE Trans. Software Eng. **41**(6), 603–619 (2015). DOI 10.1109/TSE.2014.2383381. URL https://doi.org/10.1109/TSE.2014.2383381
[78] Xu, T., Jin, L., Fan, X., Zhou, Y., Pasupathy, S., Talwadker, R.: Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 307–319 (2015)
[79] Yilmaz, C., Cohen, M.B., Porter, A.A.: Covering arrays for efficient fault characterization in complex configuration spaces. IEEE Transactions on Software Engineering **32**(1), 20–34 (2006)
[80] Zheng, W., Bianchini, R., Nguyen, T.D.: Automatic configuration of internet services. ACM SIGOPS Operating Systems Review **41**(3), 219–229 (2007)
[81] Zhou, M., Chen, Q., Mockus, A., Wu, F.: On the scalability of linux kernel maintainers' work. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, pp. 27–37 (2017)