Noname manuscript No.
(will be inserted by the editor)

Empirical Assessment of Generating Adversarial Configurations for Software Product Lines

Paul Temple · Gilles Perrouin · Mathieu Acher · Battista Biggio · Jean-Marc Jézéquel · Fabio Roli

the date of receipt and acceptance should be inserted later

Abstract Software product line (SPL) engineering allows the derivation of products tailored to stakeholders' needs through the setting of a large number of configuration options.

Unfortunately, options and their interactions create a huge configuration space which is either intractable or too costly to explore exhaustively. Instead of covering all products, machine learning (ML) approximates the set of acceptable products (*e.g.,* successful builds, passing tests) out of a training set (a sample of configurations). However, ML techniques can make prediction errors yielding non-acceptable products wasting time, energy and other resources.

We apply *adversarial machine learning techniques* to the world of SPLs and craft new configurations faking to be acceptable configurations but that are not and vice-versa. It allows to diagnose prediction errors and take appropriate actions. We develop two adversarial configuration generators on top of state-

Paul Temple
NaDI, PReCISE, Faculty of Computer Science, University of Namur
E-mail: paul.temple@unamur.be

Gilles Perrouin
NaDI, PReCISE, Faculty of Computer Science, University of Namur
E-mail: gilles.perrouin@unamur.be

Mathieu Acher
Univ Rennes, IRISA, Inria, CNRS
E-mail: mathieu.acher@irisa.fr

Battista Biggio
University of Cagliari
E-mail: battista.biggio@unica.it

Jean-Marc Jézéquel
Univ Rennes, IRISA, Inria, CNRS
E-mail: jezequel@irisa.fr

Fabio Roli
University of Cagliari
E-mail: fabio.roli@unica.it

of-the-art attack algorithms and capable of synthesizing configurations that are both adversarial and conform to logical constraints.

We empirically assess our generators within two case studies: an industrial video synthesizer (MOTIV) and an industry-strength, open-source Web-app configurator (JHipster). For the two cases, our attacks yield (up to) a 100% misclassification rate without sacrificing the logical validity of adversarial configurations. This work lays the foundations of a quality assurance framework for ML-based SPLs.

Keywords software product line; configurable system; software variability; software testing; machine learning; quality assurance

1 Introduction

> Testers don't like to break things; they like to dispel the illusion that things work. [50]

Software Product Line Engineering (SPLE) aims at delivering *massively customized* products within shortened development cycles [71,29]. To achieve this goal, SPLE systematically reuses software assets realizing the functionality of one or more *features*, which we loosely define as units of variability. Users can specify products matching their needs by selecting/deselecting the features and provide additional values for their attributes. Based on such *configurations*, the corresponding products can be obtained as a result of the product derivation phase. A long-standing issue for developers and product managers is to gain confidence that all possible products are functionally viable, *e.g.,* all products compile and run. This is a hard problem since modern software product lines (SPLs) can involve thousands of features inducing a combinatorial explosion of the number of possible products. For example, in our first case study (the MOTIV video generator), the estimated number of configurations is 10^{314} while the derivation of a single product out of a configuration takes 30 minutes on average. At this scale, practitioners cannot test all possible configurations and the corresponding products' qualities.

Variability models (*e.g.,* feature diagrams) and solvers (SAT, CSP, SMT) are widely used to compactly define how features can and cannot be combined [8,76,15,14]. Together with advances in model-checking, software testing and program analysis techniques, it is conceivable to assess the functional validity of configurations and their associated combination of assets within a product of the SPL [22,83,23,28,10,60]. In addition to the validity of configurations, their acceptability with regards to users expectations must be assessed. We faced this situation in an industrial context: despite significant engineering effort [39], the MOTIV SPL – used to generate videos that benchmark object recognition techniques – keeps deriving videos that are typically too noisy or dark. In that case, both image analysis algorithms and human experts will fail to recognize anything resulting in a tremendous waste of resources and negative user experience. To handle this issue, a promising approach is to sample a

number of configurations and predict the quantitative or qualitative properties of the remaining products using Machine Learning (ML) techniques [80, 74, 43, 11, 81, 62, 86, 84].

However, we need to trust the ML classifier [7, 61] of an SPL in avoiding misclassifications and costly derivations of non-acceptable products. ML researchers demonstrated that some forged data, called *adversarial*, can fool a given classifier [21]. *Adversarial machine learning* (advML) thus refers to techniques designed to fool (*e.g.,* [17, 16, 61]), evaluate the security (*e.g.,* [19]) and even improve the quality of learned classifiers [41]. Even though results are promising in different contexts, the ML community did not apply advML techniques in the SPL domain. On the other hand, numerous techniques have been developed to test or learn software configuration spaces of SPLs, but none of them considered advML [68]. A strength of advML is that generated adversarial configurations are crafted to force an ML classifier to make errors, by either exploiting its intrinsic properties or its insufficient training. Furthermore, since advML operates on the classifier, there is no need to derive and test additional products of an SPL.

The main idea of this article is to shift ideas and techniques from advML to the engineering of SPLs or configurable systems. Specifically, the principle is to generate adversarial configurations with the intent of fooling and improving ML classifiers of SPLs. Adversarial configurations can pinpoint cases for which non-acceptable products of an SPL can still be derived since the ML classifier is fooled and misclassifies them. Such configurations are symptomatic of issues stemming from various sources: the variability model (e.g., constraints are missing to avoid some combinations of features); the variability implementation (e.g., interactions between features cause bugs); the testing environment (e.g., some products are wrongly tested and should not be considered as acceptable); or simply the fact that, based on previous observations, configurations are predicted to meet non-functional requirements while they actually fail to do so, asking to be fixed.

In this article, our overall goal is to assess how and to what extent advML techniques can be used for ML-based SPLs. We demonstrate these techniques in a binary classification setting (acceptable/non-acceptable) where acceptability is either defined as combinations of visual properties (MOTIV SPL) or failed/successful builds. This paper makes the following contributions:

1. the development of two adversarial generators on top of state-of-the-art attack algorithms (evasion attacks) and capable of synthesizing configurations that conform to logical constraints among options;
2. the usage and applicability of our two generators within two case studies: an industrial video synthesizer (MOTIV) and an industry-strength, open-source Web-app configurator (JHipster). The two systems come from different domains (video processing *vs.* Web), variability is implemented differently (Lua code and parameters *vs.* conditional compilation over different languages), and size of the configuration space differs (10^{314} *vs.* 90K configurations);

3. the assessment of the effectiveness of the use of adversarial configurations via two research questions: *i) How effective is our adversarial generator to synthesize adversarial configurations?* We answered this question by generating adversarial configurations and confronting against a random strategy, up to 100% of the generated adversarial configurations conforms to the variability model and are successfully misclassified; and *ii) What is the impact of adding adversarial configurations to the training set regarding the performance of the classifier?* Based on our results, twenty-five adversarial configurations are sufficient to affect the classifier performance. We also provide statistical evidence supporting our results;
4. the public availability of our implementation and empirical results at `https://github.com/templep/EMSE_2020`.

This article is an extension of "Towards quality assurance of software product lines with adversarial configurations" published at SPLC 2019 (full paper, research track) [85]. In this extension, we describe the implementation of a new adversarial generator for SPLs with a new preprocessing technique on top of SecML, a Python library that implements state-of-the-art adversarial algorithms. We broaden the applicability and assessment of the dedicated algorithm we used in our previous work with a new case study (JHipster) in a different software engineering context.

This new generator produces adversarial configurations up to 50% faster than our initial algorithm [85]. Overall our empirical assessment suggests that generating adversarial configurations is effective to investigate the quality of ML-based SPLs.

The rest of this paper is organized as follows. Section 2 provides background information about SPL, ML and advML. Section 3 describes how advML can be used in the context of SPL engineering, including details about algorithms for generating adversarial configurations. Section 4 introduces our two case studies, while Section 5 describes research questions and experimental setup. Sections 6 and 7 describe our empirical results. Sections 8 and 9 present some potential threats to validity and provide qualitative insights on how SPLs practitioners can leverage adversarial configurations. Finally, Section 10 covers related work and Section 11 wraps up the paper.

2 Background

In this section, we introduce the necessary background and concepts of a machine learning-enabled software product line framework where ML is used to classify configurations of an SPL [91,53,40,68,86,84,2,6].

2.1 SPL framework

SPL engineering aims at delivering customized products out of software features' values (configurations). Figure 2 illustrates the process with one of the

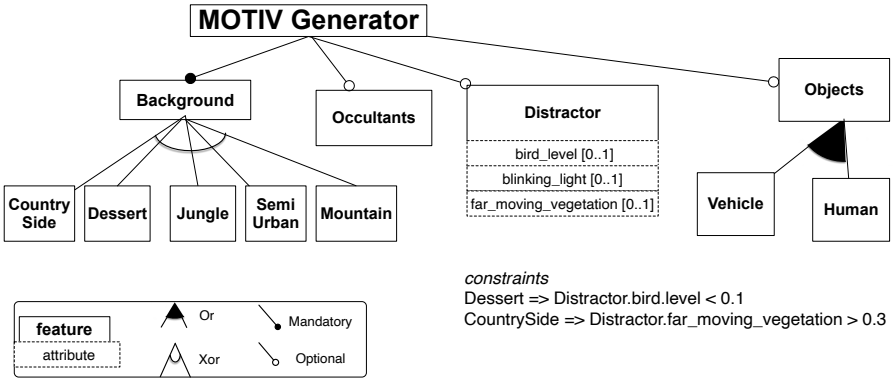


Fig. 1: Simplified MOTIV feature model [38, 4]

case studies of this article (*i.e.,* the MOTIV video generator): Products (also called variants) are videos and have been synthesized out of configuration files documenting values of features. There are three important steps in the process: variability modeling, variability implementation, and variant acceptability. We illustrate these concepts on the MOTIV case study [38, 86, 4].

Variability modeling. A *variability model* defines the features and attributes (also called configuration options) of an SPL; various formalisms (*e.g.,* feature models [51], decision models) can be employed to structure and encode information [15, 14]. A variability model typically defines domain values for each feature and attribute. Moreover, as not all combinations of values are permitted, it is common to write additional constraints regarding features and attributes (*e.g.,* mutual exclusions between two Boolean features). For example, in the feature model depicted Figure 1, we can use either CountrySide, Dessert, Jungle, SemiUrban, or Moutain to synthesize the Background but not a combination of them. The way to specify constraints and their expressiveness depends on the variability model's semantics [75]. For Boolean feature models, constraints apply to individual features (mandatory or optional), parent-child relationships (the selection of a child feature implies the selection of its parent), group of child features within a variability operator (alternative, optional). Additionally *cross-tree* constraints relate arbitrary features and attributes; there are expressed textually in the form of logical formulae. For example, enabling Dessert requires a value of *bird_ level* less than 0.1. A *configuration* is an assignment of values to every individual feature and attribute. Because of constraints and domain values, the notions of *valid* and *invalid* configurations emerge. That is, some combinations of values are accepted while others are rejected. A *constraint solver* (*e.g.,* SAT, CSP, or SMT solver) is usually employed to check the validity of configurations and reason about the configuration space of a variability model. The choice of the solver depends on the nature of the features (Boolean, numeric, etc.) and the type of constraints (hard or soft) at hand. Our case studies illustrate this diversity, MOTIV has an attributed

feature model (depicted Figure 1) for which we enumerate the valid configurations via the Choco CSP solver [38,86,4] while Jhipster's configurations have been obtained via a SAT solver [45].

Variability implementation. Configurations are only abstract representations of variants in terms of the specification of enabled and non-enabled features. There is a need to shift from the problem space (configurations) to the solution space (variants actually realizing the functionality of configurations). Different variant implementation techniques can be used such as conditional compilation (*#ifdefs* or parameters evaluated at runtime). Additionally, one can use parameters in function calls – related to values of configurations – in the generator code that ultimately produces a variant. In the MOTIV case (see Figure 2), the generator is written in Lua and uses different parameters to execute a given configuration and produce a variant. JHipster uses a questionnaire to guide users through the choice of options values and then derives automatically the desired variant (a Web development stack) – Section 4 gives more details about this subject system.

Variant Acceptability. In some cases, configurations can lead to undesirable variants despite being logically valid within the variability model. For instance, when considering the MOTIV SPL [39,85], some video variants may contain too much noise or not enough contrast. Variants can still be generated but these videos are not exploitable for any object recognition task, since the non-functional property (here: the visual quality of a video) does meet expectations. The *test oracle* is a procedure to determine whether a variant is acceptable or not in the solution space. In Figure 2, the oracle gives a label (green/acceptable or red/non-acceptable). Given a large number of variants, it is unfeasible to ask a human to assess the visual properties of all videos. Instead, we implemented a C++ procedure for computing these properties.

If a variant is considered non-acceptable by the oracle, then, there is a difference between the decision given by the solver within the problem space and the testing oracle within the solution space. Such a mismatch can occur because of the transformation from the problem space to the solution space: the variability implementation can be faulty; the oracle can be hard to automate and thus introduce approximations and/or errors; the variability model might miss some constraints. Unfortunately, the testing oracle operates over one variant at a time and cannot compute the subset of acceptable variants. Furthermore, an exhaustive derivation of all variants is not possible. Hence an approach followed by many works [91,53,40,68,86,84,2,6] is to use machine learning (ML) to approximate the set of acceptable variants through the classification of configurations.

2.2 Machine Learning (ML) Classifier

ML classification. Formally, a classification algorithm builds a function $f : X \mapsto Y$ that associates a label in the set of predefined classes $y \in Y$ with configurations represented in a feature space (noted $x \in X$). With the video

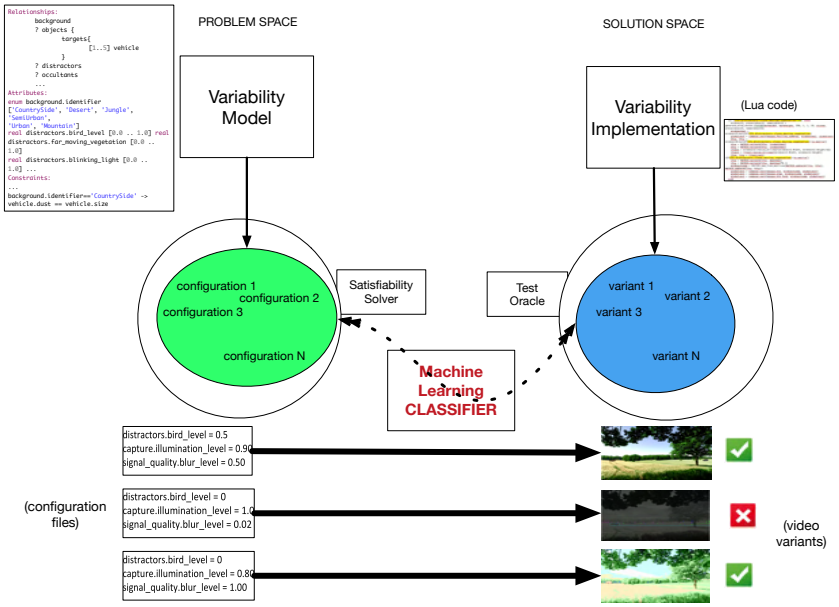


Fig. 2: Software product line and ML classifier

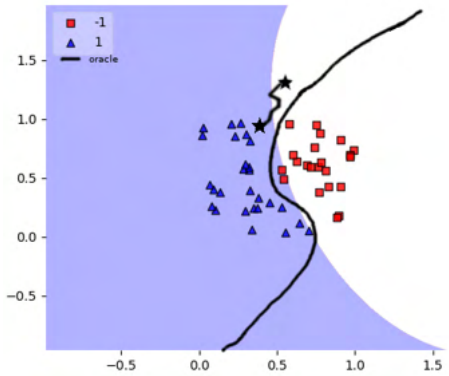


Fig. 3: Adversarial configurations (stars) are at the limit of the separating function learned by the ML classifier

generator, only two classes are defined: $Y = \{-1, +1\}$, respectively acceptable and non-acceptable videos. The configuration space X is defined by features of the underlying feature model (and their definition domain). The classifier f is trained on a data set D constituted of a set of pairs (x_i^t, y_i^t) where $x^t \in X$ is a set of valid configurations from the variability model and $y^t \in Y$ their associated labels. To label configurations in D, we use an oracle that decides on the acceptability of a configuration depending on feature values, such as luminosity and contrast. See Section 4 for implementation details of the oracle. Once the classifier is trained, f induces a separation in the feature

space (shown as the transition from the blue/left to the white/right area in Figure 3) that mimics the oracle: when an unseen configuration occurs, the classifier determines instantly in which class this configuration belongs to. Unfortunately, the separation can yield prediction errors since the classifier is based on statistical assumptions and a (small) training sample. We exemplify this in Figure 3 by the separation diverging from the solid black line representing the target oracle. As a result, two squares are misclassified as being triangles. Classification algorithms realize trade-offs between the necessity to classify the labeled data correctly, taking into account the fact that it can be noisy or biased and its ability to generalise to unseen data. Such trade-offs lead to approximations that make the classifier "weak" (*i.e.,* taking decisions with low confidence) in some areas of the configuration space. Comparatively to other domains such as hiring, loan decisions, healthcare, or security, misclassifying video sequences may not be considered as highly critical: practitioners are "only" wasting computation time and resources in generating non-acceptable videos. Yet, approximations and classification errors may have negative financial and user experience impacts. We notice similar concerns in the case of JHipster, our second case study (see Section 4). In both cases, the SPLs cannot be deployed and commercialized as such and further engineering effort is needed to improve their quality.

3 Adversarial Machine Learning and Evasion attacks

In SPL engineering, ML brings the benefit of partitioning the configuration space based on a (small) number of assessed variants, which is faster than running the oracle on every single variant (videos in the MOTIV case, see Figure 2). However, this gain comes at the cost of approximations made by the statistical ML classifier. That is, the ML classifier can still make prediction errors when classifying configurations (see Figure 3). Our idea is to "attack" the ML classifier through the generation of so-called adversarial configurations able to fool the ML classifier of an SPL. The objective is to synthesize configurations for which the ML classifier performs an inaccurate classification. For example, such adversarial configurations can pinpoint which non-acceptable variants of an SPL can still be derived since the ML classifier misclassifies them as acceptable.

In this section, we detail the foundations, algorithms, and processes to generate adversarial configurations.

3.1 AdvML and evasion attacks

According to Biggio *et al.* [21], deliberately attacking an ML classifier with crafted malicious inputs was proposed in 2004. Today, it is called adversarial machine learning and can be seen as a sub-discipline of machine learning. Depending on the attackers' access to various aspects of the ML system (*e.g.,*

access to the data sets or ability to update the training set) and their goals, various kinds of attacks are available [17,20,16,19,18]. A categorization of such adversarial attacks can be found in [7,21]. In this paper, we focus on *evasion attacks*: these attacks move labeled data to the other side of the separation (putting it in the opposite class) via successive modifications of features' values. Since areas close to the separation are of low confidence, such adversarial configurations can have a significant impact if added to the training set. To determine in which direction to move the data such that it reaches the separation, a gradient-based method has been proposed by Biggio *et al.* [16]. This method requires the attacked ML algorithm to be differentiable (*e.g.*,, algorithms building models for which the classification decision is based on a confidence metric which is not binary; this is the case for SVMs or Bayesian predictors which compute a likelihood to belong to a class). One of such differentiable classifiers is the Support Vector Machine (SVM), parameterizable with a kernel function[1]. Note also that, in the context of this work, we focus on a binary classification problem, but the framework presented by Biggio *et al.* [21] applies in a broader case, including multi-class problems.

3.2 A dedicated evasion algorithm

Algorithm 1 A dedicated algorithm [85] conducting the gradient-descent evasion attack inspired by [16]

Input: x^0, the initial configuration; t, the step size; nb_disp, the number of displacements; g, the discriminant function
Output: x^*, the final attack point

(1) m = 0;
(2) Set x^0 to a copy of a configuration of the class from which the attack starts;
while $m < nb\_disp$ **do**
  (3) m = m+1;
  (4) Let $\nabla F(x^{m-1})$ a unit vector, normalisation of $\nabla g(x^{m-1})$;
  (5) $x^m = x^{m-1} - t\nabla F(x^{m-1})$;
end while
(6) return $x^* = x^m$;

Algorithm 1 presents an adaptation of Biggio *et al.*'s evasion attack [16], initially presented at the SPLC 2019 conference [85]. First, we select an initial configuration to be moved (x^0); multiple strategies can be used to select x^0, we will simply use a random strategy selecting one configuration labeled with the class from which the attack starts. Then, we set the step size (t), a parameter controlling the convergence of the algorithm. Large steps induce difficulties to converge, while small steps may trap the algorithm in a local optimum. While the original algorithm introduced a termination criterion based on the impact of the attack on the classifier between each move (if this impact was smaller

[1] most common functions are linear, radial-based functions and polynomial

than a threshold ϵ, the algorithm stopped; assuming an optimal attack), we chose to set a maximal number of displacements nb_disp in advance and let the technique run until the end. This allows for a controllable computation budget, as we observed that for small step sizes the number of displacements required to meet the termination criterion was too large. The function g is the discriminant function (*i.e.,* the function that should be differentiable) and is defined by the ML algorithm that is used. It is defined as $g : X \mapsto \mathbb{R}$ that maps a configuration to a real number. Only the sign of g is used to assign a label to a configuration x. Thus, $f : X \mapsto Y$ can be decomposed in two successive functions: first $g : X \mapsto \mathbb{R}$ that maps a configuration to a real value and then $h : \mathbb{R} \mapsto Y$ with $h = sign(g)$. However, $|g(x)|$ (the absolute value of g) intuitively reflects the confidence the classifier has in its assignment of x. $|g(x)|$ increases when x is far from the separation and surrounded by other configurations from the same class and is smaller when x is close to the separation.

The term discriminant function has been used by Biggio *et al.* [16] and should not be confused with the unrelated discriminator component of generative adversarial nets (GANs) by Goodfellow *et al.* [41]. In GANs, the discriminator is part of the "robustification process". It is an ML classifier striving to determine whether an input has been artificially produced by the other GANs' component, called the generator. Its responses are then exploited by the generator to produce increasingly realistic inputs. In this work, we only generate adversarial configurations, though GANs are envisioned as follow-up work.

Concretely, the core of the algorithm consists of the *while* loop that iterates over the number of displacements. Statement (4) determines the direction towards the area of maximum impact with respect to the classifier (explaining why only a unit vector is needed). $\nabla g(x^{m-1})$ is the slope of the gradient of $g(x^{m-1})$. Since evasion attacks is a technique based on gradient descent, the direction of interest towards which the adversarial configuration should move is the opposite of this value. This vector is then multiplied by the step size t and subtracted to the previous move (5). The final position is returned after the number of displacements has been reached. For statements (4) and (5) we simplified the initial algorithm [16]: we do not try to mimic as much as possible existing configurations as we look forward to some diversity. In an open-ended feature space, the gradient can grow indefinitely possibly preventing the algorithm to terminate. Biggio *et al.* [16] set a maximal distance representing a boundary of the feasible region to keep the exploration under control.

In SPLs, the feasible region is given by valid configurations (defined by, among others, allowed features' combinations). However, being able to state all cross-tree constraints and potential domain values remain difficult. This task is nonetheless very important for the adversarial attack algorithm. In this work, we opted for a quite simplistic way of handling constraints. We only took care of the type of features and attribute values (natural integers, floats, Boolean). For example, if a constraint forbids a value to go below zero but a displacement tries to do so, we reset to zero this value (since it is the lower

bound that this value can take); a similar principle is done for Boolean values (that can take only values 0 or 1).

Temple *et al.* [86] studied the possibility to use ML to discover previously unstated constraints in a VM and add them in the model. These constraints relate to the definition of acceptable products and the idea of being able to only derive products that satisfy users' requirements. This work results in an automated process to specialize SPLs to meet these requirements. We used decision trees since they were very well adapted to the context of the study as they are interpretable and constraints can be retrieved by going through the structure of the resulting tree. However, in the context of advML, not all ML models can be used and decision trees are not compatible [21,19,17,16]. Decision trees are highly non-linear and it is not possible to compute gradient nor confidence due to non-differentiability. To conduct our attack, we choose to use support vector machines instead. Note that this restriction only applies to conduct adversarial attacks, when trying to specialize an SPL, any ML model can be used.

3.3 secML

SecML[2] is a Python library that has been developed by researchers from the Pattern Recognition and Applications Laboratory (PRALab), in Sardinia, Italy. SecML has been publicly released for the first time the 6^{th} of August 2019[3]. This library gathers different advML techniques and embeds utilities to create a customized pipeline according to the data to attack, their representations, the ML model that is used in the system to attack among other parameters. SecML was designed as a generic advML library but was not tailored to analyze classifiers for SPLs. An interesting question was about the effort of adapting secML algorithms in this novel application domain, in particular regarding constraints. As further motivation, we want to compare how the original advML algorithm behaves compared to our aforementioned dedicated algorithm [85].

SecML offers different implementations of adversarial attacks[4], either poisoning the training set, trying to evade the classifier, or even being completely customized. For each category, different implementations are also proposed. For instance, regarding evasion attacks, multiple implementations are provided[5], some can hide even more implementations, such as the Cleverhans attack which is based on an external library[6] providing even more possibilities[7]. Among all of them, CAttackEvasionPGD is probably the most direct imple-

[2] `https://secml.gitlab.io/index.html`

[3] Therefore it was not available in our previous SPLC'19 contribution [85]

[4] `https://secml.gitlab.io/tutorials.adv.html`

[5] `https://secml.gitlab.io/secml.adv.attacks.evasion.html`

[6] `https://secml.gitlab.io/secml.adv.attacks.evasion.html#module-secml.adv.attacks.evasion.cleverhans.c_attack_evasion_cleverhans`

[7] `https://secml.gitlab.io/tutorials/09-Cleverhans.html`

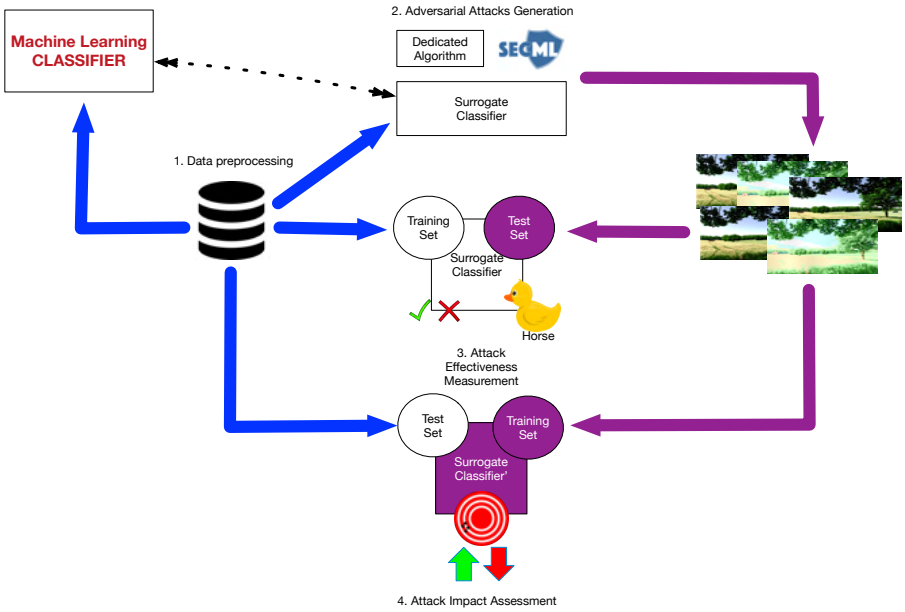


Fig. 4: Adversarial Pipeline

mentation of the algorithm presented in [16], proposing the evasion attack. Therefore, we focus our attention on this implementation for our experiments. PGD refers to Projected Gradient Descent which allows limiting the maximum amount of perturbations that can be applied to a configuration. That is, if a perturbation would move the configuration outside of the defined boundaries, it is automatically set back on these boundaries via projection. The maximal amount of perturbations is defined by the parameter called d_max that we will set to different values in our experiments (see Section 6 and 7).

3.4 Adversarial pipeline

Figure 4 presents a generic adversarial pipeline. The first step is to prepare the data that are shared by the original classifier and by the adversarial pipeline. Data preprocessing is specific to each case and is described in Sections 6 and 7. Generally, an adversarial framework relies on a *surrogate classifier* that is learned from the same data when the attacker does not have access to the target classifier or when the attack cannot be conducted directly. Since there is evidence that attacks conducted on a specific ML model can be transferred to others [33,32,24], using a surrogate classifier is a legit approach in a black-box scenario.

Our experiments are conducted within a white-box scenario: we have access to all the SPL artifacts including the ML classifier. Therefore, the surrogate and the original SPL classifiers conflate and, without loss of generality, we can use a differentiable classifier. Attacks will be conducted and assessed on that

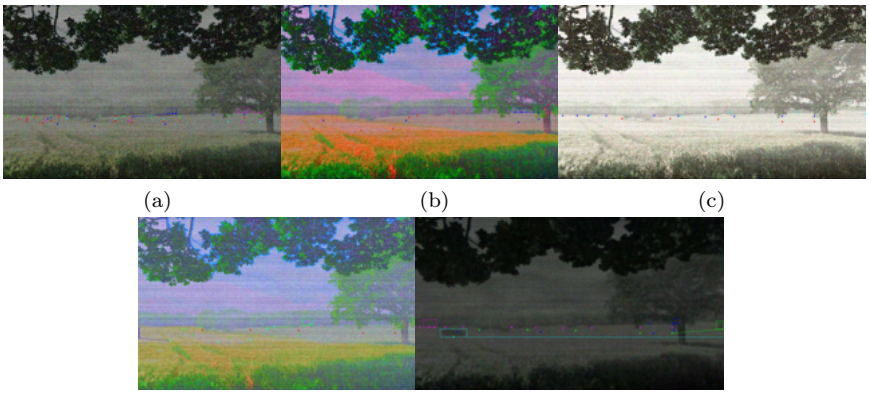
unique classifier. Once the classifier is learned, we can use our dedicated and SecML algorithms to generate attacks in a second step. The third step evaluates the effectiveness of generated adversarial configurations forming the test set. In particular, we check the validity of generated adversarial configurations and their ability to be misclassfied. This constitutes our first research question. Finally, the fourth step learns a new classifier with an augmented training set composed of the original training set and some adversarial configurations. Our second research question assesses the positive or negative impact on the classifier's accuracy.

4 Case Studies

4.1 MOTIV video generator

MOTIV is an industrial video generator of which the purpose is to provide synthetic videos that can be used to benchmark computer vision based systems. Video sequences are generated out of configurations specifying the content of the scenes to render [86,85]. MOTIV relies on a variability model that documents possible values of more than 100 configuration options, each of them affecting the *perception* of generated videos and the achievement of subsequent tasks, such as recognizing moving objects. Perception's variability relates to changes in the background (*e.g.,* forest or buildings), objects passing in front of the camera (with varying distances to the camera and different trajectories), blur and other combinations of elements such as camera movements, ambient daylight or fog. There are 20 Boolean options, 46 categorical (encoded as enumerations) options (*e.g.,* to use predefined trajectories) and 42 real-value options (*e.g.,* dealing with blur or noise). On average, enumerations contain about 7 elements each and real-value options vary between 0 and 27.64 with a precision of 10^{-5}. Excluding (very few) constraints in the variability model, we overestimate the video variants' space size: $2^{20} * 7^{46} * ((0 - 27.64) * 10^5)^{42} \approx 10^{314}$. Concretely, MOTIV takes as input a text file describing the scene to be captured by a synthetic camera as well as recording conditions. Then, we run Lua [46] scripts to compose the scene and apply desired visual effects resulting in a video sequence. To realize variability, the Lua code uses parameters in functions to activate or deactivate options and to take into account values (enumerations or real values) defined into the configuration file. A highly challenging problem is to identify feature values and interactions that make the identification of moving objects extremely difficult if not impossible. Typically, some of the generated videos contain too much noise or blur. In other words, they are *not acceptable* as they cannot be used to benchmark object tracking techniques. Another class of non-acceptable videos is composed of the ones in which the same value is given to all pixels of every frame, resulting in a succession of still images: nothing can be perceived.

Figure 5 shows some examples of non-acceptable videos that can be generated with MOTIV. On these images, there is noise preventing human beings



(a) (b) (c)

(d) (e)

Fig. 5: Examples of non-acceptable generated videos

to perceive the background and whether a vehicle is present and their identifications. In addition, for Figures 5a, 5c and 5e, contrast is poor while Figure 5b and 5d present unrealistic colors.

Non-acceptable videos represent a waste of time and resources: 30 minutes of CPU-intensive computations per video on average, without including the time to run benchmarks related to object tracking (several minutes depending on the computer vision algorithm). We therefore need to constraint our variability model to avoid such cases.

Previous work. We previously used ML classification techniques to predict the acceptability of unseen video variants [86]. We summarise this process in Figure 6. We first sample valid configurations using a random strategy (see Temple *et al.* [86] for details) and generate the associated video sequences. Our testing oracle labels videos as acceptable (in green) or non-acceptable (in red). This oracle implements image quality assessment [35] defined by the authors via an analysis of frequency distribution given by Fourier transformations. An ML classifier (in the case of [86], a decision tree) can be trained on such labelled videos. "Paths" (traversals from the root to the leaves) leading to non-acceptable videos can easily be transformed into new constraints and injected in the variability model.

AdvML to the rescue. An ML classifier can make errors, preventing acceptable videos (false positives) or allowing non-acceptable videos (false negatives). Most of these errors can be attributed to the confidence of the classifier coming from both its design (*i.e.,* the set of approximations used to build its decision model) and the training set (and more specifically the distribution of the classes). Areas of low confidence exist if configurations are very dissimilar to those already seen or at the frontier between two classes. In the reminder, we investigate the use of advML to quantify these errors and their impact on MOTIV SPL and ML classifier.

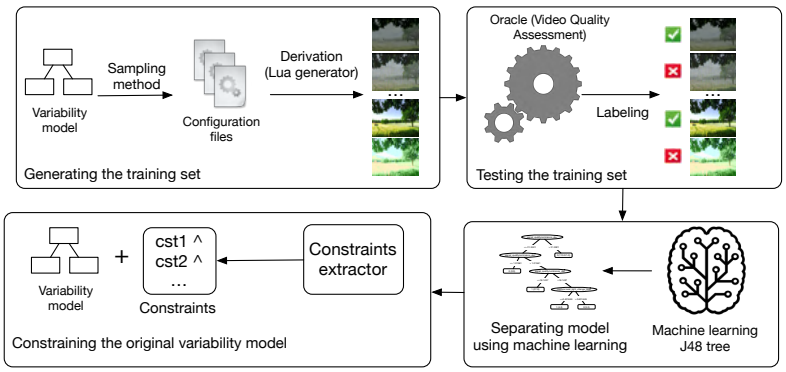


Fig. 6: Refining the variability model of the MOTIV video generator via an ML classifier. In this article and as a follow up of this engineering effort, our goal is to generate adversarial configurations capable of fooling the ML classifier.

4.2 JHipster

JHipster is an open-source generator for developing Web applications [47]. Started in 2013, the JHipster project is popular with more than 15,000 stars on GitHub and gathers a strong community of more than 500 contributors in December 2019. JHipster is used by many companies and governmental or research organisations worldwide, including Google, Ericsson, CERN or the Italian Research Council (CNR).

From a user-specified configuration, JHipster generates a complete technological stack constituted of Java and Spring Boot code (on the server side) and Angular and Bootstrap (on the front-end side). The generator supports several technologies ranging from the database used (e.g., *MySQL* or *MongoDB*), the authentication mechanism (e.g., *HTTP Session* or *Oauth2*), the support for social log-in (via existing social networks accounts), to the use of microservices. Technically, JHipster uses *npm* and *Bower* to manage dependencies and Yeoman[8] (aka *yo*) tool to scaffold the application [73]. JHipster relies on conditional compilation with EJS[9] as a variability realisation mechanism. The mechanism is similar to #ifdef with CPP preprocessor and is applied on different files written in different languages: Java, JavaScript, CSS, Docker files, Maven or Gradle files. The build process resolves variability scattered in numerous files and is quite costly (10 minutes on average per configuration).

Previous work. We previously used JHipster as a case study to benchmark sampling techniques and assess their bug-finding effectiveness [45,69]. Lessons learned from our study are that building a configuration-aware testing infrastructure for JHipster requires a substantial effort both in terms of human and computational resources. Specifically, we relied on 8 man-months for building the infrastructure and 4376 hours of CPU time as well as 5.2 terabytes of disk

[8] `http://yeoman.io/`
[9] `https://ejs.co/`

space used to build and run all JHipster configurations. Another lesson is that our exhaustive exploration of JHipster variants is not practically viable.

AdvML to the rescue. Instead of deriving all variants, one can use ML and only a sample of configurations to eventually prevent non-acceptable variants and avoid costly build. Such effort can also be exploited as part of the continuous integration of JHipster. The process of Figure 6, illustrated for MOTIV, is conceptually similar for JHipster. We have a feature model documenting the possible configurations and materialized as configuration files. The variants (or products) are not videos this time, but variants of source code written in different languages. As an outcome, we can identify features of JHipster that cause non-acceptable variants (*i.e.,* build failures) and re-inject this knowledge into the feature model. Build failures can occur in various circumstances such as: (1) implementation bugs in the artefacts, typically due to a dependency wrongly specified in a Maven file or due to unsafe interactions between features in the Java source code; (2) un-properly building environments in which some packages or tools are incidentally missing because some combinations of features were not assessed before. Once the learning process of Figure 6 is realized, the question arises as to the quality of the ML classifier and the whole JHipster SPL. Again, we can apply advML.

In this article, we use the version 4.8.2 whose reverse-engineered feature model is available online[10]. The feature model allows one to build all JHipster configurations. Yet, in our sampling we made a few restrictions to focus on the most relevant ones. In particular, we selected all the testing frameworks (*Gatling*, *Cucumber*, *Protractor*) in each sampled configuration and avoided configurations that required Oracle to focus on non-proprietary variants. This feature model allows 90,210 variants in total.

4.3 Cases Synthesis

Table 1 summarises the main characteristics of the case studies analysed in this work. We notice that the domain greatly influences constraints stated in the variability model. While JHipster exhibits a high Cross-Tree Constraints Ratio (CTCR), proactively preventing the majority of build failures, MOTIV allows any combination of features, sometimes leading to unacceptable videos (soft constraints). From an adversarial machine learning perspective, MOTIV raises the challenge of navigating in a huge variability space with an imperfect oracle implementing visual perception. JHispter is requiring to handle constraints, specifically when running an evasion attack, since the risk of generating useless adversarial configurations is very high if we ignore them.

[10] `https://github.com/templep/EMSE_2020`

Subject	Domain	Nb. Features	Nb. Variants	Feature Type	CTCR (%)	Task
MOTIV	Video	108	10^{314}	Boolean, Categorical, Real	0% (0/108)	Visual Acceptability classification
JHipster	Web development	58	90210	Boolean	53% (31/58)	Passed/Failed build classification

Table 1: Case studies' characteristics. We report for each case the domain, number of features and variants, the type of features, the Cross-tree constraints Ratio (CTCR) that is the number of features involved in cross-tree constraints to the total number of features and finally the goal of the classsification task.

5 Evaluation Overview

In this section, we introduce the research questions related to the use of advML attacks for SPLs and present how we implemented them. The last part describes how we preprocessed data and the parameterization of the techniques used to run our experiments.

5.1 Research Questions

We address the following research questions:

RQ1: *How effective is our adversarial generator to synthesize adversarial configurations?* Effectiveness is measured through the capability of our evasion attack algorithm to generate misclassified configurations:

- **RQ1.1**: Can we generate adversarial configurations that are wrongly classified?
- **RQ1.2**: Are all generated adversarial configurations valid w.r.t. constraints in the VM?
- **RQ1.3**: Is using the evasion algorithm more effective than generating adversarial configurations with random modifications?
- **RQ1.4**: Are attacks effective regardless of the targeted class?

RQ2: *What is the impact of adding adversarial configurations to the training set regarding the performance of the classifier?* The intuition is that adding adversarial configurations to the training set could improve the performance of the classifier when evaluated on a test set.

We answer these questions for each case study separately in Sections 6 and 7. First, we state the evaluation protocol and then the results to these answers. We answer each question using both techniques presented in Sections 3.2 and 3.3. We also provide statistical evidence on the fact that our results are significantly different from results without using any advML technique. To do so, we performed Mann-Whitney tests as detailed in the evaluation protocol (see Section 5.4)

5.2 Implementation

We implemented the dedicated algorithm in Python 3 (scripts are available on the companion website).

MOTIV's variability model embeds enumerations which are usually encoded via integers. The main difference between the two is the logical order that is inherent to integers but not encoded into enumerations. As a result, some ML techniques have difficulties to deal with them. The solution is to "dummify" enumerations into a set of Boolean features, without forgetting to enforce inherent exclusion constraints of literals from the original enumerations. Conveniently, Python provides the *get_dummies* function from the *pandas* library which takes as input a set of configurations and feature indexes to dummify.

For each feature index, the function creates and returns a set of Boolean features representing the literals' indexes encountered while running through the set of given configurations: if the *get_dummies* function detects values in the integer range $[0, 9]$ for a feature associated to an enumeration, it will return a set of 10 Boolean features representing literals' indexes in that range. The function takes care of preserving the semantics of enumerations. However, dummification is not without consequences for the ML classifier. First, it increases the number of dimensions: our 46 initial enumerations would map to 145 features that may expose the ML algorithm to the *curse of dimensionality* [13]; as the number of features increases in the feature space, configurations that look alike (*i.e.,* with close feature values and the same label) tend to go away from each other, making the learning process more complex. This curse has also been recognized to have an impact on SPL activities [31]. Second, dummification implies that we will operate our attacks in a feature space *essentially different* from the one induced by the real SPL. Thus, we need to transpose the generated attacks in the dummified feature space back to the original SPL one, raising one main issue: there is no guarantee that an attack relevant in the dummified space is still efficient in the reduced original space (the separation may simply not be the same). For instance, dummification will break a categorical feature into multiple independent binary features. The attack will process each feature individually, it may result that two binary features to be activated at the same time as they are considered independently one from another. The original categorical feature implicitly encodes a dependency between all the available literals not reported by the dummification procedure and thus both feature space (*i.e.,* before and after the transformation) are not equivalent. Further efforts are required to ensure these implicit constraints.

Additionally, exclusion constraints stated in the FM and enumerations become non-correlated after dummification allowing gradient methods to operate on each feature independently. That is, when transposed back to the original configuration space, invalid configurations would need to be "fixed", potentially putting these adversarial configurations away from the optimum computed by the gradient method. In the following, we will only perform transformations on

the initial feature space that can be reversed (*e.g.,* normalize values between 0 and 1) to conduct our evasion attacks, the transition between the two feature spaces (*i.e.,* initial and after preprocessing) is possible, thus, we do not make any further distinctions between the two terms since we use them without making any transformations.

As mentioned in Section 3.2, we conducted attacks on a SVM with a linear kernel which is a simple classifier (*i.e.,* only linear separation can be created) but that performs already very well on the classification task for both case studies. Scripts as well as data used to compare predictions can be found on the companion webpage.

5.3 Presentation of the results

In sections 6 and 7, we present the results of our experiments for each RQs stated in Section 5.1. We try to keep the same structure for each addressed RQ, except for RQ 1.3 which is the comparison to random perturbations and does not use either of the implementations. First, we describe the preprocessing applied to the data and the parameterization of the two different implementations of the evasion attack. Then we address each RQ by describing the intent and details about how we conducted our experiments and discussing the results that come with an illustration. Finally, we provide some insights based on these results. We repeat the same schema for the second implementation.

5.4 Evaluation protocol

Data collection: Previous work on both case studies [86, 44, 70] allowed one to gather a number of configurations (*i.e.,* 90, 210 configurations from JHipster in its version 4.8.2. and 4, 500 randomly sampled and valid video configurations for MOTIV). Configurations were sampled, derived (executed or build), and assessed using a computing grid.

Training and Test sets: In the 4, 500 MOTIV configurations, about 10% are non-acceptable (see Section 4.1). Because of the vast majority of acceptable configurations, we are not able to use common ML practices: usually, the training set is composed of a high percentage (*e.g.,* around 66%) of available data and is used to train the classifier, and, when few configurations are available, k-fold cross-validation is used to mitigate the risk of overfitting (in our case, 4, 500 configurations is an arguably low number to the size of the variant space). Regarding the number of configurations to put in the training set, 66% of 4, 500 configurations would reduce the size of the test set in turns containing very few non-acceptable configurations. In such a setting, because of the approximations explained in Section 3.3, the classifier can easily learn a function that will not separate anything while keeping good classification performances since non-acceptable configurations would be considered as "rare events" in numbers. Because of that, we decided to mitigate this risk by training a classifier with even fewer configurations but keeping the same ratio of

non-acceptable configurations (*i.e.,* $\approx 10\%$) leading us to 500 configurations in the training set among which around 50 are non-acceptable. Results from [86] showed good results in terms of classification on both training and test sets which made us keep this setting.
Now, regarding cross-validation, it is used to validate/select a classifier when several are created (*e.g.,* when fine-tuning hyper-parameters) but it requires separating the training set into smaller subsets. With 10% of non-acceptable configurations some subsets will not contain any of them which is counter-productive when assessing the ability of a trained classifier to successfully classify new configurations.

In this setting, the key point is that only about 10% of configurations are non-acceptable. This is a ratio that we cannot control as it depends on the targeted non-functional property. However, to reduce imbalance, several data augmentation techniques exist like SMOTE [27]. Usually, they create artificial configurations while maintaining the configurations' distribution in the feature space. We decided to follow a similar process by computing the centroid between two configurations (of the same class) and use this point as a new configuration.

Thanks to the centroid method, we can bring a perfect balance between the two classes (*i.e.,* 50% of acceptable configurations and non-acceptable configurations). Technically: we compute the number of configurations needed to have perfectly balanced sets (*i.e.,* for both training and test sets); we select randomly two configurations from the less represented class and compute the centroid between them, check that it is a never-seen-before configuration and add it to the available configurations. The process is repeated until the required number of configurations is reached. Once a centroid is added to the set of available configurations, it is available as a configuration to create the next centroid.

In the following, we present results with both original and balanced data sets to assess the impact of class representation on adversarial attacks.

On the other hand, JHipster presents about $20,000$ configurations in the dataset that cannot be built. This represents about nearly 25% of our dataset, which shows an over-representation of the building class. While other problems (described later on under the label **Feature space structure**) remain, this representation is still an issue and we had to force sets to be perfectly balanced. Thus, we chose to reduce the training set to 400 configurations. In this training set, both classes are represented equally (*i.e.,* 200 configurations for each class). That is, we choose randomly 200 configurations from the ones that build and 200 other configurations from the ones that do not.

Preprocessing: We have applied some preprocessing to both datasets and configurations' representations to remove unnecessary features gathered from our previous study. For instance, for JHipster, we removed logs that were kept for further studies to retrieve the root causes of bugs. They were stored as free text which prevented us to have an homogeneous representation of the configurations in terms of features. Furthermore, error messages from the logs are not necessary for this study. If a feature only reports one value, it is

discarded since it will only increase the number of dimensions in the feature space without adding any information to the classification problem.

Since we use SVMs to conduct evasion attacks, we need to make our feature space homogeneous in its dimension. Integer values must be scaled-down between 0 and 1. The same applies to floating-point values. The reason behind this decision is to avoid unbounded values which in turn can result in bigger importance given to a feature because it is more flexible and can take a broader range of values. To cope with homogeneous feature spaces, we applied dummification (see Section 5.2) to categorical features. This way, every single literal of the feature becomes an independent Boolean feature.

After these preprocessing steps, MOTIV configurations are represented by about 120 features while JHipster's configurations contain 47.

Feature space structure: Regarding JHipster, our preliminary study, to assess that at least one ML model was able to classify configurations between building or not building, showed that this classification problem is linearly separable. Even with as few as $\approx 18,000$ configurations (about 20% of available configurations) in the training set, the classifier as able to perform more than 99% of correct prediction in the test set. Below this number, our tentatives to learn a classifier with a random selection of configurations have often failed due to the absence of non-building configurations. With more than 99% accurate classifications, adversarial attacks were unable to produce any configurations that were misclassified by the classifier. Besides, unlike the MOTIV case study, after our feature transformation, JHipster presents numerous dependencies between the features (*i.e.,* the choice of using a SQL database force to deselect any other kind of database) making the adversarial attacks unlikely to produce any valid configurations regarding the associated feature model. We add to enforce encoded constraints (choices, dependencies stated as constraints in the FM and upper and lower bounds values) in the FM directly into the attack procedure. That is, after the last displacement has been performed, we check that all the constraints are fulfilled and, if not, modify crafted configurations accordingly. This way, configurations produced by the adversarial procedure are valid by design. This is also known as the "fix operator" in configuration optimization [65].

Parameterization of the techniques: We configured the dedicated attack generator with the following settings: *i)* we set the number of attacks points to generate $4,000$ configurations for RQ1 with MOTIV and $1,000$ when dealing with JHipster and, for RQ2, 25 configurations were used for both case studies; *ii)* considered step size (t) values are $\{10^{-6}; 10^{-4}; 10^{-2}; 1; 10^{2}; 10^{4}; 10^{6}\}$; *iii)* the number of iterations is fixed to 20, 50 or 100. To mitigate randomness, we repeat ten times the experiments. All results discussed in this paper can also be found on our companion webpage[11].

When using secML, we have parameterized the attack function such that it can change several features at one time (distance norm is set to "l2" also known as Euclidean distance), the maximum perturbation value (called d_max here-

[11] `https://github.com/templep/EMSE_2019`

after) is set to $\{0.1; 0.5; 1.0; 5.0; 10.0\}$ and the upper and lower bounds are set respectively to 1 and 0. Also, Gradient Descent uses a search grid to find the best step size as well as the best direction which can be configured by the resolution of the grid. In secML, this parameter is called η that we set to 0.01 since the order of magnitude of the number of features is 10^2. To remain consistent with our dedicated implementation, we apply the same preprocessing on the configurations (see Section 5.2) and we generate 4000 configurations for RQ1 and 25 configurations when addressing RQ2.

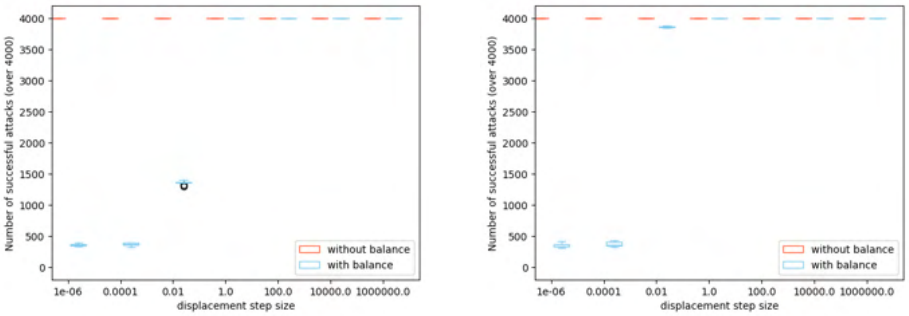
Statistical Evidence: To further support the statistical validity of our conclusions we applied statistical tests for each research question. We focused on Mann-Whitney rank sum test [58], which is non-parametric (*i.e.,* it does not assume a normal distribution for the compared samples) and therefore can handle the non-normality of our sometimes small sample sizes. This test computes a statistic U, further decomposed in U_1 and U_2 that intuitively correspond to the number of wins and losses respectively of all pairwise contests between the two compared samples. Therefore, $0 < U_i < n_1 \times n_2, i \in \{1, 2\}$ where n_1, n_2 are the sizes of each sample and $U_1 + U_2 = n_1 \times n_2$. We report $min(U_1, U_2)$ referred to as "u-stat". For this test, we formulate two different hypotheses: the null-hypothesis (usually called H0) that supposes the two samples follow the same distributions and the other hypothesis (H1) supposing that they do not. To reject the null-hypothesis, u-stats are compared to a critical value reported in a table[12]: if a u-stat is smaller or equal to the one in the table for the given sample sizes and confidence levels (in our case $\alpha = 0.05$), then we can reject H0, if it is greater then we cannot reject H0. Additionally, two values of u-stat are remarkable: u-stat = 0, meaning that the distributions are totally different, and u-stat = $\frac{n_1 \times n_2}{2}$ meaning that the two distributions are equal. In our setting, we use a two-tailed test and additionally report the computed p-value p for significant (we consider our results significant if $p \leq 0.05$). We computed our statistical results using the R statistical environment [72]. The R script to replicate the analysis can be found on our companion webpage.

6 MOTIV

6.1 RQ1: How effective is our adversarial generator to synthesize adversarial configurations?

To answer this question, we assess the number of wrongly classified adversarial configurations over 4000 generations and compare them to a random baseline: to the best of our knowledge, there is no other evasion attack that is based on a different algorithm than the one presented in [16] to compare to.

[12] Such tables can be found easily on the Internet: `http://ocw.umb.edu/psychology/psych-270/other-materials/RelativeResourceManager.pdf`



(a) Number of misclassified adversarial configurations (20 displacements)

(b) Number of misclassified adversarial configurations (100 displacements)

Fig. 7: Number of successful attacks on class *acceptable*; X-axis represents different step size values t while Y-axis is the number of misclassified adversarial configurations by the classifier. For each t value, results with balanced and not balanced training set are shown (respectively in blue and orange).

6.1.1 RQ1.1: Can we generate adversarial configurations that are wrongly classified?

Dedicated implementation
For each run, after *nb_disp* displacements, the newly created adversarial configuration is added to the set of initial configurations that can be selected to start an evasion attack. We thus allow previous adversarial configurations to continue their displacements towards the global optimum of the gradient.

Figure 7 shows box-plots resulting from ten runs for each attack setting. Both results, when the training set is not balanced (*i.e.,* using the previous training set containing 500 configurations with about 10% of non-acceptable configurations) and when it is balanced (*i.e.,* increasing the number of non-acceptable configurations using the data augmentation technique described above) are reported.

Both Figure 7a and Figure 7b indicate that we can always achieve 100% of misclassified configurations with our attacks. Regarding Figure 7a, in the case of a not balanced dataset, it is easy to attack the most represented class and the implementation can produce new configurations that are always misclassified (regardless of the value assigned to t). The fact that even the smallest value of t with 20 displacements can produce 4000 misclassified configurations suggests that the separation learned by the classifier should be very close to the configurations of class *non-acceptable* (where the attacks start). With a balanced dataset, more points can be selected as a starting point of the attack. Chances to select a starting point that is "far" from the separation are higher and more displacements are needed to cross the function learned by the classifier. Figure 7a shows that the displacement step size is not enough

to produce any misclassified configurations when it is set to a value lower or equal to 10^{-4} and the number of displacement is set to 20. When set to 0.01, some configurations start to be misclassified but it represents less than 50% of them. With a step size set to 1.0 or above, perturbations are large enough to produce misclassified ones.

Figure 7b shows the same tendency except that the transition from 0 misclassified generations to 4000 appears earlier (*i.e.,* when t is set to or close to 0.01). It is not surprising since, compared to the previous results, the number of displacements is higher (set to 100).

We do not present figures with the number of displacements set to 50 but the observations are similar to the ones we just described with Figure 7b. However, they can be found on the companion webpage.

Table 2 shows U (second row) from the Mann-Whitney test and associated p-values (third row) when comparing results from both sets (balanced and unbalanced) for each displacement step size shown in Figure 7a. When the displacement step size is set to 1 or higher, both attack set-ups achieve a 100% success rate with no dispersion (illustrated by 50 and NaN values in the table) and therefore cannot be distinguished. With lower values of displacement step sizes, the p-values are significant ($p << 0.05$). Based on our results from Figure 7a, we can conclude that attacking when classes are not balanced is easier. Conclusions are similar when considering larger displacement step size (*i.e.,* 50 and 100) available on the companion page.

	10^{-6}	10^{-4}	10^{-2}	1	10^2	10^4	10^6
u-stat	0	0	0	50	50	50	50
p-value	$6.39e-5$	$6.39e-5$	$6.39e-5$	NaN	NaN	NaN	NaN

Table 2: u-statistics and p-values associated with measures from Fig. 7a; $n_1 = n_2 = 10$ for each column, u-stat$_{critical} = 23$.

Insights: Increasing the number of displacements requires lower step sizes to reach the misclassification goal but it comes at the cost of more computations. However, increasing the number of displacements when the step size is already large ends up in incredibly large displacements which may not be realistic in some applications or when trying to limit changes applied to configurations.

SecML

Similarly to the previous implementation, after the final perturbation is applied to a configuration, it is added to the set of available configurations that may be selected for the next run. The number of displacements is not bounded directly, thus the number of iterations can vary from one run to another.

Figure 8 shows box-plots resulting of ten runs for each attack setting (*i.e.,* with varying values set to d_max). We also show results for both balanced and non-balanced data sets.

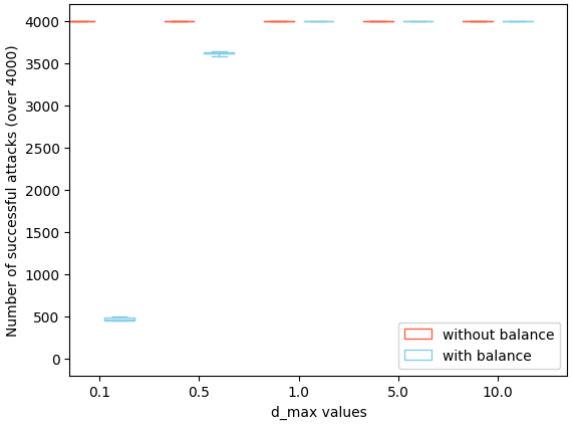


Fig. 8: Number of successful attacks on class *acceptable*; X-axis represents different parameter values of *d_ max* of the secML function while Y-axis is the number of misclassified adversarial configurations by the classifier.

When the data set is not balanced, similarly to the previous implementation, all attempts to create a misclassified configuration succeed since the separation is in favor of the most-represented class.

When the data set is balanced, the behavior showed in Figure 8 is similar to the previous one. A transition from 500 misclassified generated configurations to almost 4000 can be seen between d_max set to 0.1 and 0.5. With higher d_max values, all 4000 generated configurations were misclassified.

Our statistics (available on Table 3) shows that our results are significant ($p << 0.05$). Again, with this implementation, we can conclude that attacking classes is easier when they are imbalanced.

	0.1	0.5	1	5	10
u-stat	0	0	12.5	12.5	12.5
p-value	$7.49e-3$	$7.49e-3$	NaN	NaN	NaN

Table 3: u-statistics and p-values associated with measures from Fig. 8; $n_1 = n_2 = 5$ for each column, $\text{u-stat}_{critical} = 2$.

Insights: again tuning d_max increases chances to get misclassified configurations when perturbed with evasion attacks. Since d_max represents the maximum distance up to which a feature can be perturbed, setting it a lower value decreases the potential number of iterations to get to the final position. Therefore, to fine-tune this parameter, a good strategy would be to start with lower values.

6.1.2 RQ1.2: Are all generated adversarial configurations valid w.r.t. constraints in the VM?

As discussed in Section 5.2, we have performed some preprocessing on our data to make the learning with an SVM possible and reliable. This ends up with features that are either Boolean, either real but bounded between 0 and 1.

Because we check and force feature values to be inside these boundaries after the last position of a configuration is reached, by design, all the configurations are valid w.r.t. this aspect. The only aspect left that may make the configurations non-valid is the mutual exclusion constraints inherited from breaking the categories with the dummification process.

Regardless of the implementation that is used or the values given to parameters, all the generated configurations are valid. Results and scripts of this experiment can be found on the companion website.

Insights: We can scope parameters such that adversarial configurations are *both successful and valid* for either implementations. The way our configurations were preprocessed made it possible to enforce boundary constraints directly at the end of displacements. Other ways are possible but might require other mechanisms to check that constraints are verified while allowing configurations to move further away.

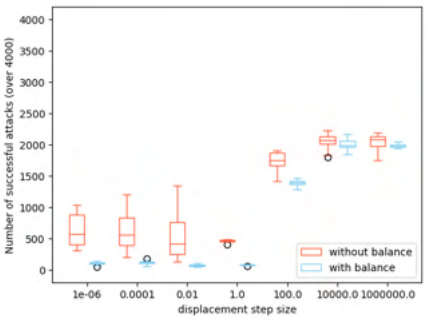
6.1.3 RQ1.3: Is using the evasion algorithm more effective than generating adversarial configurations with random displacements?

Previous results of RQ1.1 and RQ1.2 show we can craft valid adversarial configurations that can be misclassified by the ML classifier but is our algorithm better than a random baseline?

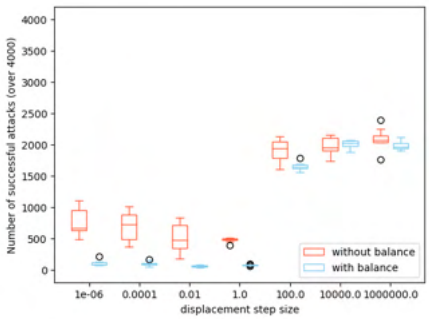
The baseline algorithm is based on the dedicated implementation and consists in: i) for each feature, choose randomly whether to modify it; ii) choose randomly to follow the slope of the gradient or go against it (the role of '-' of line 5 in Algorithm 1 that can be changed into a '+'); iii) choose randomly a degree of displacement (corresponding to the slope of the gradient ($\nabla F(x^{m-1})$) of line 5 in Algorithm 1). Both the step size and the number of displacements are the same as in the previous experiments.

Figure 9 shows the ability of random attacks to successfully mislead the classifier. Random modifications are not able to produce more than 2500 configurations that are misclassified (regardless of the number of displacements, the step size, or whether the training set is balanced or not) which corresponds to about 60% of the generated configurations. It is a lower number than the two evasion implementations. The maximum number of misclassified configurations after random modifications starts from step size $t = 10,000$ regardless of the studied number of displacements.

Regarding the validity of generated configurations, here again, the random version is worse than the other considered implementations. The problem lies



(a) Number of successful random attacks after 20 displacements



(b) Number of successful random attacks after 100 displacements

Fig. 9: Number of successful random attacks on class *acceptable*; X-axis represents different step size values t while Y-axis is the number of misclassified adversarial configurations by the classifier. In red and blue are respective results with a not balanced and a balanced training set in terms of classes representation.

in the fact that all features are processed independently from the others resulting in high chances to set to features which are mutually exclusive to 1; leading to non-valid configurations.

Table 4 shows the u-statistics and p-values when comparing results given from the RQ1.1 (see Figure 7a) and random perturbations (see Figure 9a) with balanced classes. For all reported values, the p-values are significant ($p << 0.05$) and U-stats are equal to 0. We conclude that advML techniques are more efficient in producing new configurations that will be wrongly classified than using random perturbations. Results are similar when comparing results from Figure 7b and Figure 9b.

	10^{-6}	10^{-4}	10^{-2}	1	10^{2}	10^{4}	10^{6}
u-stat	0	0	0	0	0	0	0
p-value	$1.82e-4$	$1.82e-4$	$1.82e-4$	$6.34e-5$	$6.34e-5$	$6.39e-5$	$6.39e-5$

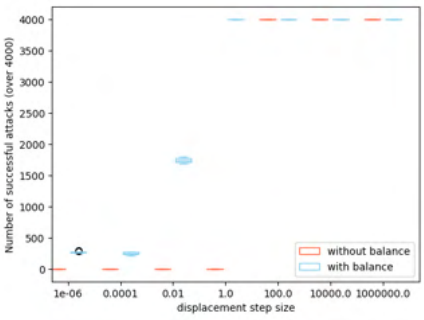
Table 4: u-statistics and p-values associated with measures from Fig. 9a compared with values given in RQ1.1 when classes are balanced; $n_1 = n_2 = 10$ for each column, u-$stat_{critical} = 23$.

Insights: Previous results show that the effectiveness of evasion attacks are superior to random modifications since i) evasion attacks can craft configurations that are always misclassified by the ML classifier while less than 2500 over 4000 generations will be misclassified using random modifications; ii) generated evasion attacks support a larger set of parameter values for which generated configurations are valid; iii) we were able to identify sweet spots for

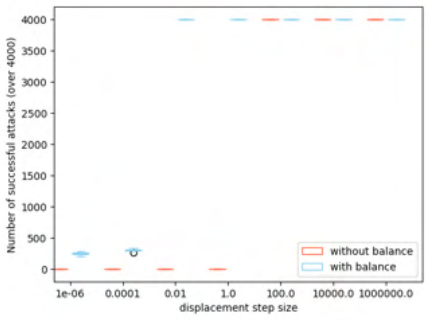
which evasion attacks were able to generate 4000 configurations that were both misclassified and valid.

6.1.4 RQ1.4: Are attacks effective regardless of the targeted class?

Previously, we generated evasion attacks from the class *non-acceptable* and tried to make them acceptable for the ML classifier but is our attack symmetric? Now, we configure both adversarial configuration generators so that they move configurations from the *acceptable* class to the class *non-acceptable*.



(a) Number of successful adversarial attacks after 20 displacements



(b) Number of successful adversarial attacks after 100 displacements

Fig. 10: Number of successful adversarial attacks on class *non-acceptable*; X-axis represents different step size values t while Y-axis is the number of misclassified adversarial configurations by the classifier; In orange and blue are respectively shown results when the training set is not balanced and when it is.

Dedicated implementation

Figure 10a shows that, in the case of balanced data sets, all generated configurations are misclassified when step size is set to 1 or higher with 20 displacements while, when this number is 100 (see Figure 10b), the step size can be set to 0.01 or higher. With t set to 0.01, a transition occurs in Figure 10a in the number of misclassified configurations from 1800 to 4000. The same transition probably occurs also in Figure 10b but more abruptly and thus is hidden between observations made at $t = 10^{-4}$ and $t = 0.01$. The fact that almost or no configurations are misclassified before this threshold comes from the over-representation of class *acceptable*. Since there are more configurations assigned to this class, there are more candidates to start an attack including configurations that are far away from the separation learned by the classifier.

Table 5 compares results when attacking class *acceptable* and when attacking class *non-acceptable* with 20 displacements allowed and when classes are

balanced. When the displacement step size is set to 1.0 or higher, both achieve a 100% success rate showing no difference in the results (represented by NaN in the table). When the displacement step size is set to lower values, p-values are significant ($p << 0.05$). We conclude that it is easier to fool the classifier when starting attacks from the under-represented class (here "non-acceptable", see RQ1.1).

	10^{-6}	10^{-4}	10^{-2}	1	10^{2}	10^{4}	10^{6}
u-stat	0	0	0	50	50	50	50
p-value	$1.83e-4$	$1.83e-4$	$1.82e-4$	NaN	NaN	NaN	NaN

Table 5: u-statistics and p-values associated with measures from Fig. 10a compared with results from Fig. 7a for displacement step size set to 20; $n_1 = n_2 = 10$ for each column, u-$stat_{critical} = 23$.

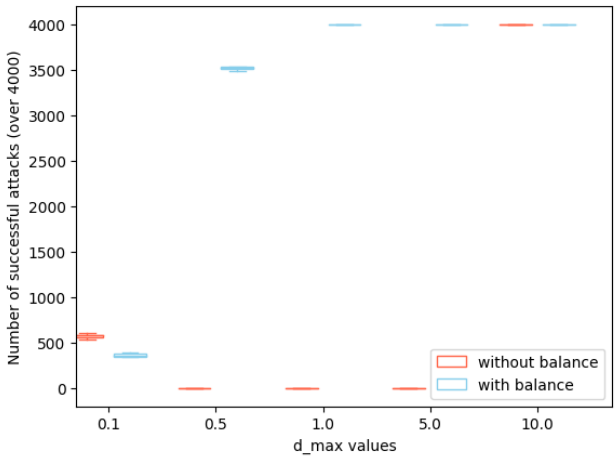


Fig. 11: Number of successful attacks on class *non-acceptable*; X-axis represents different parameter values of d_max of the secML function while Y-axis is the number of misclassified adversarial configurations by the classifier.

SecML

Figure 11 shows the number of misclassified configurations when the evasion attacks target class *non-acceptable*. While in the balanced case, results are similar to the one we described with the dedicated implementation; results for non-balanced datasets are more interesting. When d_max is set to a value that is under 10.0, no generated configurations are misclassified[13]. This is related

[13] except when d_max is set to 0.1 and for which we do not have any explanation

to over-representation issues and as secML implements some mechanisms to stop computations early, attack configurations do not move away.

From the validity point of view, the same implementations were used as well as the checking procedure. All generated configurations are valid regardless of the parameterizations of the implementations or the choice of the implementation. Results are provided on the companion webpage.

Table 6 reports statistical results when comparing values given in Figure 8 and Figure 11 with balanced classes. Once again, p-values show that our results are significantly different ($p << 0.05$). Only the last column shows that results are the same which translate into a 12.5 u-statistics and a NaN p-value.

	0.1	0.5	1	5	10
u-stat	0	0	0	0	12.5
p-value	$7.49e-3$	$3.98e-3$	$3.98e-3$	$3.98e-3$	NaN

Table 6: u-statistics and p-values associated with measures from Fig. 8 compared with the ones from Fig. 11; $n_1 = n_2 = 5$ for each column, u-$\text{stat}_{critical} = 2$.

Our generated adversarial attacks are: 100% effective (always misclassified, RQ1.1), do not depend on the target class (RQ1.4), and yield valid configurations (RQ1.2) if parameterized properly. In contrast, our random baseline was only able to achieve 62.5% of effectiveness at best (RQ1.3). The balance in the data sets does not affect these results and the targeted class affects show the same trends despite small differences (RQ1.4).

6.2 **RQ2: What is the impact of adding adversarial configurations to the training set regarding the performance of the classifier?**

So far, we have only evaluated the impact of generated attacks and how they are predicted by the classifier. Yet, some ML techniques (*e.g.,* GANs) take advantage of adversarial instances by incorporating them in the training set to improve the classifier confidence and possibly performance. In our context, we want to assess the impact of our attacks when they are included in the training dataset, especially with less "aggressive" (*e.g.,* small step sizes and a low number of displacements) configurations of the attacks.

Dedicated implementation
To do so, we allowed 20 displacements to avoid configurations moving too far from their initial positions but used the same step sizes that we have used before. For each step size, we generate 25 adversarial configurations (targeting the acceptable class) that are added all at once in the training set, we retrain

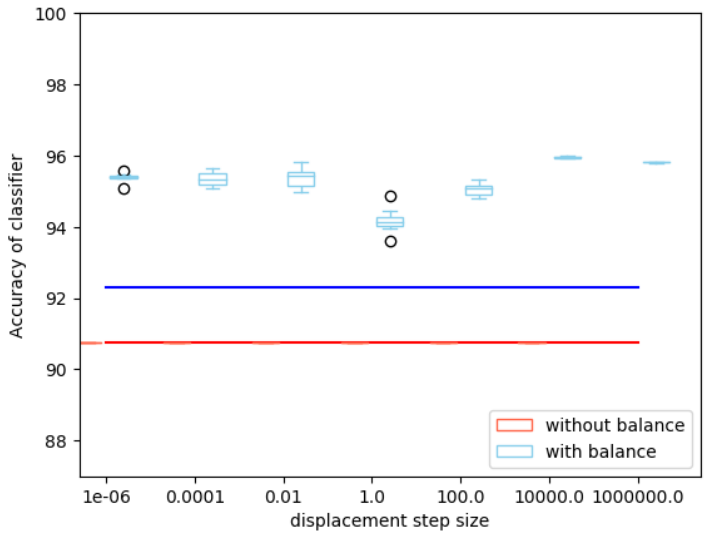


Fig. 12: Accuracy of the classifier after retraining with 25 adversarial configurations in the training set over a test set of 4000 configurations (7000 configurations when the training set is balanced). In red are results when no balance are forced in the classes, in blue, both training set and test set are balanced. The initial accuracy of the classifier is represented by the horizontal line (90.7661% for the red line and 92.3151% for the blue one). X-axis represents different step size values t while Y-axis is the accuracy of the classifier (zoomed between 87% and 100%).

the classifier and evaluate it on the configurations that constitute the initial test set (without any adversarial configurations in it). Every retraining process was repeated ten times to mitigate the effects of the random configuration selection. We also present results when the training set is balanced, in which case we have also augmented the test set to bring balance and to follow the same data distribution. In this case, the test set does not contain 4000 configurations but about 7000 in which 50% of the configurations are considered acceptable and the remaining are considered non-acceptable.

Figure 12 shows the accuracy of the retrained classifiers over a test set composed of 4000 configurations for the red part and 7000 configurations for the blue one.

The initial accuracy of the classifier was 90.7661% over the same 4000 configurations and is shown as the horizontal red line. We make the following observation: our generated configurations did not have any impact on the retraining since our boxplots are completely flat and superimposed with the baseline. We suppose that to have an impact, a larger number of adversarial configurations would be needed but with the risk of making the prediction performances of the classifier worse.

In the case of balanced training and test sets (in blue on Figure 12), results are completely different. Regardless of the value assigned to t, prediction performances after retraining are above the baseline (*i.e.,* 92.3151%). The gain is in between +1.5% and +2%. While this gain seems low, we remind that initial performances were already very high (above 90%) which makes it difficult to retrieve a huge improvement.

Statistics in Table 7 show that, probably due to the presence of outliers and large variance in the boxplots of Figure 19, with displacement values lower than 1, results are not necessarily significant while, above, they are (p << 0.05).

	10^{-6}	10^{-4}	10^{-2}	1	10^{2}	10^{4}	10^{6}
u-stat	40	30	40	0	0	0	0
p-value	0.69	0.20	0.69	8.46e − 5	8.54e − 5	8.18e − 5	7.59e − 5

Table 7: u-statistics and p-values associated with measures from Figure 12 comparing results from the balanced experiment with associated baseline; $n_1 = n_2 = 10$ for each column, u-stat$_{critical} = 23$.

SecML

With 25 adversarial configurations added to the training set, results presented in Figure 13 are rather different from the other implementation. Considering the lower red part of the Figure (*i.e.,* without balance), when d_max is set to 0.1, the predictions of the retrained classifier can gain up to 2% but when the parameter is set to higher values, the performances drop. It can due to adversarial configurations going deep inside the other class which might make ultimately more harm than good.

When data sets are balanced, the prediction performances tend to drop when d_max is set to 1.0 or lower values. If d_max is set to 5.0 or higher, performances tend to slightly increase. Again, initial performances are already high (*i.e.,* above 95%) and we have only added 25 configurations to a training set containing about 1000 configurations which can limit the impact of adversarial configurations on the potential improvement of performances.

Table 8 shows results for both retraining (in a balanced setting and imbalanced setting). In the balanced setting, only the first column shows non-significant results ($p >> 0.05$), the remaining columns show a clear confidence in our results. Our analysis of Figure 13 still holds, we cannot state strongly that what we observed is true but the fact that the baseline is already high and that we are able to see variations in our measures suggest that retraining may still have an impact on the classifier. In the unbalanced setting, only d_max set to 0.1 provides significant results, other results report non-significant p-values ($p > 0.05$). Figure 13 shows that, except when d_max is set to 0.1, the set of measures are spread above or below the baseline. Sometimes only outliers are above the red baseline, which may explain such statistics.

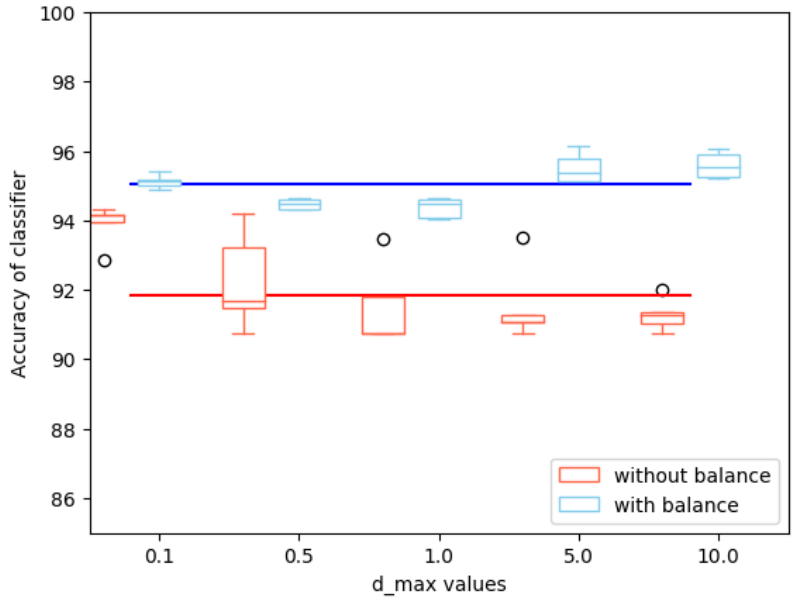


Fig. 13: Accuracy of the classifier after retraining with 25 adversarial configurations in the training set over a test set of 4000 configurations (7000 configurations when the training set is balanced). In red are results when no balance are forced in the classes, in blue, both training set and test set are balanced. The initial accuracy of the classifier is represented by the horizontal line (90.7661% for the red line and 95.1196% for the blue one). X-axis represents different d_max values while Y-axis is the accuracy of the classifier (zoomed between 87% and 100%).

	0.1	0.5	1	5	10
Balanced classes					
u-stat	10	0	0	0	0
p-value	0.66	$7.29e-3$	$7.49e-3$	$7.29e-3$	$7.49e-3$
Unbalanced classes					
u-stat	0	10	5	5	5
p-value	$7.49e-3$	0.66	0.11	0.12	0.12

Table 8: u-statistics and p-values associated with measures from Fig. 13; $n_1 = n_2 = 5$ for each column, u-stat$_{critical} = 2$.

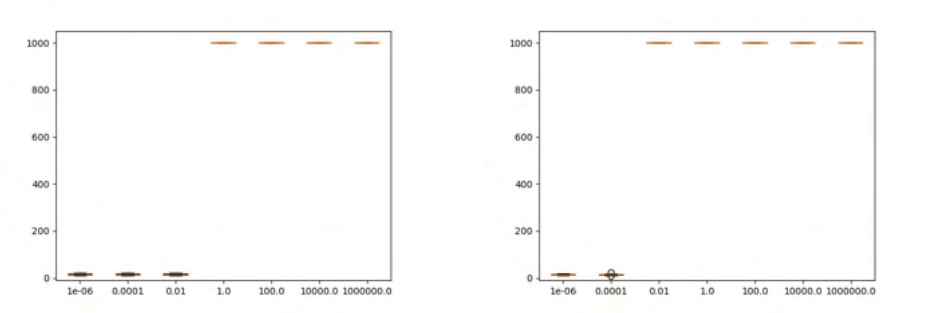
When data sets are balanced, configurations generated by evasion attacks can be used and added to the training set to improve the prediction performances of the classifier but requires empirical tuning. With only 25 configurations added, we can improve classifier accuracy by up to 3%.

7 JHipster

7.1 RQ1: How effective is our adversarial generator to synthesize adversarial configurations?

7.1.1 RQ1.1: Can we generate adversarial configurations that are wrongly classified?

Dedicated Algorithm
Similarly to MOTIV, for each of the 10 runs, after *nb_ disp* are performed, the newly generated configuration is added to the set of configurations that can be selected to start the next evasion attack giving a chance to the previously generated configurations to keep being modified and further move towards the global optimum of the gradient. Figure 14 shows box-plots summarising ten runs for the different step sizes t values we have considered.



(a) Number of misclassified adversarial configurations (20 displacements)

(b) Number of misclassified adversarial configurations (100 displacements)

Fig. 14: Number of successful attacks on class *building*; X-axis represents different step size values t while Y-axis is the number of misclassified adversarial configurations by the classifier.

Figure 14a and Figure 14b both show that we can always find a set of parameter values resulting in a misclassification of all generated configurations. Regarding Figure 14a, all generated configurations become misclassified when step size t is set to 1.0 or higher. With *nb_ disp* set to 100 (see Figure 14b), all configurations are misclassified when t is set to 10^{-2} or higher. Similar results can be obtained when the number of maximum displacements is set to 50 where all the configurations are misclassified with t set to 10^{-2} or higher.

Insights: Again, increasing the number of displacements requires lower step sizes to reach the misclassification goal. However, increasing the number of

displacements means increasing the number of iterations and the number of computations.

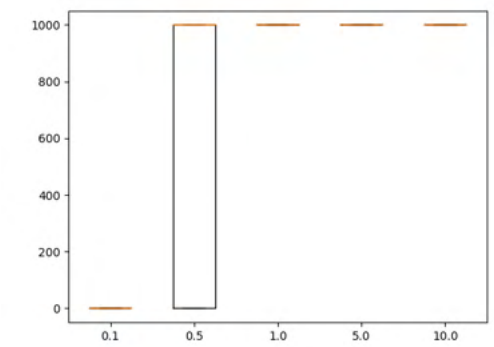
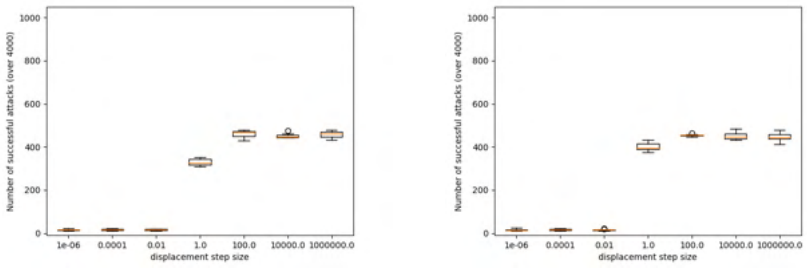


Fig. 15: Number of successful attacks on class *building*; X-axis represents different d_max values while Y-axis is the number of misclassified adversarial configurations by the classifier.

SecML
Similarly to the previous results, Figure 15 shows that we can find a value of d_max for which all generated configurations are misclassified (*i.e.,* 1.0 or above). When d_max is set to 0.1, perturbations are not sufficient for configurations to be misclassified. However, when d_max is set to 0.5, some of the runs end up with all configurations either misclassified or not which suggests that we spotted the value making the transition between 0 misclassification to 1000.

7.1.2 RQ1.2: Are all generated adversarial configurations valid w.r.t. constraints in the VM?

As we said previously, because of all the dependencies between individual features (*e.g.,* choosing the database), we had to enforce these constraints directly within the attack (*i.e.,* after the final position of the generated configuration is reached with the dedicated implementation; after each move of the configuration when using secML) otherwise, all configurations are likely to be non-valid w.r.t. the underlying variability model. Regardless of the implementations, we have enforced these constraints stated in the VM leading to valid configurations by design.



(a) Number of successful adversarial attacks after 20 displacements

(b) Number of successful adversarial attacks after 100 displacements

Fig. 16: Number of successful adversarial attacks on class *building*; X-axis represents different step size values t while Y-axis is the number of misclassified adversarial configurations by the classifier; In orange and blue are respectively shown results when the training set is not balanced and when it is.

7.1.3 RQ1.3: Is using the evasion algorithm more effective than generating adversarial configurations with random displacements?

We have used the same implementation of the random displacements that we used with MOTIV since the dedicated algorithm is the same. Unsurprisingly, results with JHipster are similar to the ones retrieved with MOTIV. Figure 16 shows the misclassification results of the generation of randomly perturbed configurations. In Figure 16a, the best results are retrieved when the step size t is set to 100 or higher. About half generated configurations are misclassified. When the number of displacement *nb_disp* is set to 100 (Figure 16b), the same conclusions can be drawn.

Table 9 reports u-statistics and p-values when comparing results from Figure 16a and Figure 14a. When the displacement step size is lower than 1, p-values are not significant ($p > 0.05$). Thus, we cannot say that using adversarial machine learning provides better results than random perturbations. In the meantime, no misclassifications were observed when using either technique. However, when displacements step size is set to 1.0 or higher, p-values are significant ($p << 0.05$) making us conclude that advML are more effective than random perturbations when these values are used. Statistics comparing Figure 16b and Figure 14b are available on the companion webpage and lead us to similar conclusions.

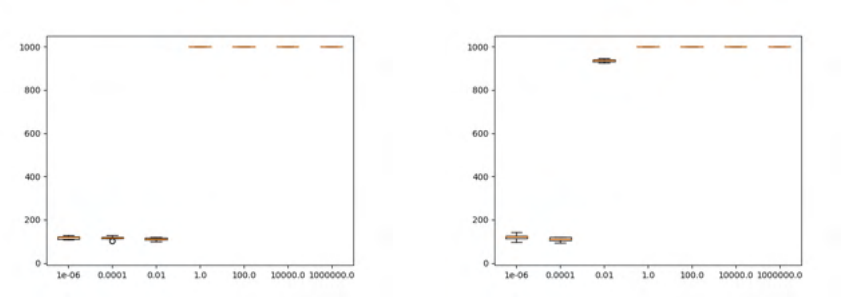
Insights: Compared to results provided by both the dedicated implementation and the secML implementation, for which we found a configuration of attacks were all generated configurations were misclassified, we can conclude that random perturbations are not as efficient with a maximum rate of 50% of misclassification over all the generations.

	10^{-6}	10^{-4}	10^{-2}	1	10^{2}	10^{4}	10^{6}
u-stat	39	44	43	0	0	0	0
p-value	0.42	0.67	0.62	$6.39e-5$	$6.39e-5$	$6.16e-5$	$6.39e-5$

Table 9: u-statistics and p-values associated with measures from Figure 16a compared from measures from Figure 14a; $n_1 = n_2 = 10$ for each column, u-stat$_{critical} = 23$.

7.1.4 RQ1.4: Are attacks effective regardless of the targeted class?

As we previously mentioned, the training set is completely balanced which should not favor the effectiveness of the attack regarding the class that it is attacking. We change the class from which attacks start (*i.e.,* going from *not building* to *building*) and assess the fact that results are similar to RQ1.1.



(a) Number of misclassified adversarial configurations (20 displacements)

(b) Number of misclassified adversarial configurations (100 displacements)

Fig. 17: Number of successful attacks on class *not building*; X-axis represents different step size values t while Y-axis is the number of misclassified adversarial configurations by the classifier.

Dedicated implementation
Figure 17 shows the effectiveness of the attack running with the dedicated implementation. It shows very similar results compared to Figure 14. In Figure 17a, starting from a step size t set to 1 and up to 10^6, all generated configurations are misclassified. Before that, the means of the box-plots are a bit higher, around 100 configurations misclassified which is still very low. The same pattern can be observed in Figure 17b compared to Figure 14b. A gap in the number of misclassified configurations can be observed when t is set to 0.01 or higher compared to lower values. Despite the fact that the same threshold can be observed in Figure 14b, the number of generated configurations that are misclassified in Figure 17b is lower that in the previous case. With t set to

lower values than 0.01, the number of generated configurations misclassified is higher than in Figure 14b but still low.

Table 10 compares measures when attacking the building class with attacks on the not building class. Measures and box-plots showed in Figure 17a and Figure 14a follow the same tendency. When the displacement set size is set to 1 or higher values, no differences can be observe explaining the reported NaN values. When set to lower values, while both numbers show low misclassification, p-values are significant ($p << 0.05$). It can be explained by the fact that Figure 14a show a capability of producing misclassified configurations close to 0 while Figure 17a shows numbers closer to 100. In this case, attacks starting from the class *building* may have more chances to succeed.

	10^{-6}	10^{-4}	10^{-2}	1	10^2	10^4	10^6
u-stat	0	0	0	50	50	50	50
p-value	$1.79e-4$	$1.81e-4$	$1.79e-4$	NaN	NaN	NaN	NaN

Table 10: u-statistics and p-values associated with measures from Figure 17a compared to measures from Figure 14a; $n_1 = n_2 = 10$ for each column, u-stat$_{critical} = 23$.

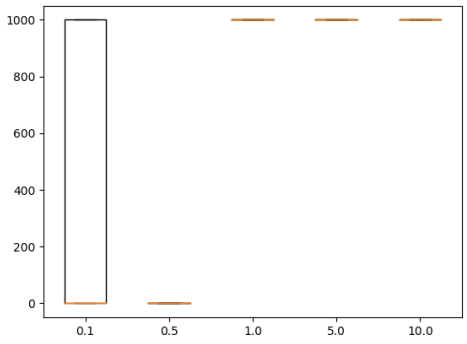


Fig. 18: Number of successful attacks on class *not building*; X-axis represents different d_max values while Y-axis is the number of misclassified adversarial configurations by the classifier.

SecML

Figure 18 shows very similar results compared to the ones described in RQ1.1. A transition from 0 misclassified configuration to 1000 can be seen when d_max is set to 1.0. However, when d_max is set to 0.5 none of the con-

figurations are misclassified. When d_max is set to 0.1, sometimes none of the configurations are misclassified, in some cases, all of them are misclassified.

Table 11 compares results when trying to attack one class or the other with the secML algorithm. Results do not show significant statistical differences (p> 0.05). This can be explained by the boxplots that spread from 0 to 1000 but with averages (*i.e.,* colored bar which is supposed to be in the middle of the box) that are either at 0 either at 1000.

	0.1	0.5	1.0	5	10
u-stat	7.5	5	12.5	12.5	12.5
p-value	0.18	0.07	NaN	NaN	NaN

Table 11: u-statistics and p-values associated with measures from Figure 18 compared from measures from Figure 15; $n_1 = n_2 = 5$ for each column, u-stat$_{critical} = 2$.

Insights: As expected, since the dataset is balanced, the behavior of both methods does not drastically change regarding the class to attack.

Here again, with JHipster, the two implementations of the evasion attack are able to generate configurations that are systematically misclassified (after tuning the parameters of the implementations).

7.2 **RQ2: What is the impact of adding adversarial configurations to the training set regarding the performance of the classifier?**

So far, only the capability of evasion attacks to generate misclassified configurations without affecting the classifier model has been assessed. Now, we are wondering whether we can include some of these generated configurations into the training set in order to improve the classification performances of the model. As stated previously, we include 25 adversarial configurations in the training set, retrain the classifier, and assess its classification performances on the initial test set (*i.e.,* without any adversarial configurations in it). Again, to avoid attacks to be too aggressive which may result in a drop in the accuracy, we limit the number of displacement for the dedicated implementation to 20. To mitigate randomness, we repeated the experiment ten times.

Dedicated implementation
As reported in Section 4.2, JHipster is a relevant case to study in this new context because it is completely unrelated to multimedia processing as opposed to historical usage of advML. JHipster has a complex structure that includes cross-tree constraints making the feature space hard to define. However, we needed to reduce drastically the number of configurations in the training set: for a large number of samples, the classifier was too good and achieved almost

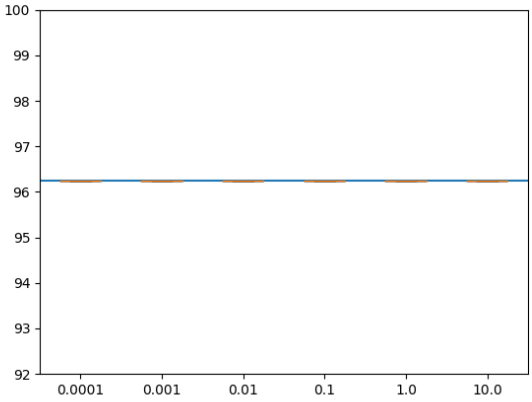


Fig. 19: Accuracy of the classifier after retraining with 25 adversarial configurations in the training set over a test set of about 89,000 configurations. The blue line reports the baseline accuracy of the classifier when no adversarial configurations are added to the training set. X-axis represents different step size values t while Y-axis is the number of misclassified adversarial configurations by the classifier.

perfect predictions. With about 400 configurations in the training set, Figure 19 shows that the generated adversarial configurations have no impact on the accuracy of the classifier after retraining. Regardless of t, all our box-plots are superimposed to the baseline. Two explanations come to mind: i) 25 might not be enough to have an impact on the function learned by the classifier, but adding more adversarial configurations may degrade the performances of the classifier; ii) the accuracy is too high without any adversarial configurations and the number of configurations in the test set is too large to capture the impact of 25 adversarial configurations in the training set.

SecML

Results showed in Figure 20 are very similar to the one described with the other implementation. With d_max set to 0.1, 0.5, or 1.0, no changes can be seen compared to the baseline. With d_max set to 5.0, the mean of the box plot stays at the same level than the baseline but some repetitions succeeded in improving the accuracy of the classifier up to the level of the baseline reported with the other implementation[14]. When d_max is set to 10.0, most of the executions (since the mean value of the box plot is above the baseline) were

[14] Note that the baselines are reported for two different models; secML provides a complete library which comes with its own framework and pipeline, necessitating to learn a classifier with this library. The implementation can differ from the ones provided by scikit-learn which is the other library we have used before using secML.

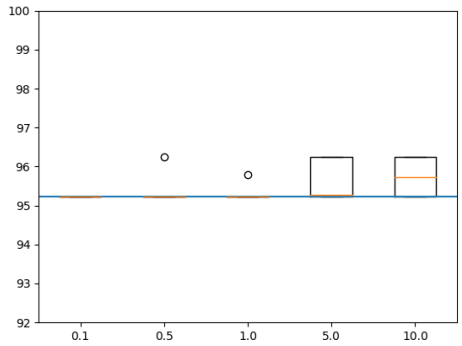


Fig. 20: Accuracy of the classifier after retraining with 25 adversarial configurations in the training set over a test set of about 89,000 configurations. The blue line reports the baseline accuracy of the classifier when no adversarial configurations are added to the training set.

able to improve the performances of the classifier up to the same level as the baseline of the previous experiment.

Table 12 suggests that the impact of retraining the classifier with adversarial examples is significant ($p < 0.05$) when d_max values is set to 5 or higher. However, the u-stat suggest that this difference is not big enough to formally reject the null hypothesis.

	0.1	0.5	1.0	5	10
u-stat	50	45	45	25	25
p-value	NaN	0.37	0.37	0.01	0.01

Table 12: u-statistics and p-values associated with measures from Figure 20 compared with the baseline; $n_1 = n_2 = 10$ for each column, u-stat$_{critical} = 23$.

Most of our attempts to improve the performance of the classifier failed since the accuracy remained the same as the baseline accuracy (*i.e.*, without adversarial configurations). This may be due to the low number of adversarial configurations that we added to the training set or it may be the nature of the classification problem that is too easy and achieving performance improvements is tricky.

8 Threats to validity

8.1 Internal threats

Choices of parameter values for our experiments may constitute a threat. The step size has been set to different powers of 10, we only used 3 different number of allowed displacements (*i.e.,* 20, 50 and 100). From our perspective, using step size below 10^{-6} in a highly dimensional space seems ridiculously small while, on the contrary, using step size above 10^{6} are tremendously large which motivates our choice to not going over these boundaries. Still, boundaries could have been extended which might have affected results regarding *RQ2* but also we could have performed more observations by using intermediate values between every power of tens. We could have provided a finer explanation of our results but at the cost of unacceptable computation times. Regarding the number of displacements, we could have used finer-grained values. We sought a compromise between allowing a lot of small steps and a few big steps. Regarding the choice of evasion attacks, as presented in Section 2, several techniques exist. Regarding secML, we set the η parameter to only one value and used five different values for d_max. While these choices were driven by the size of the feature space, we would need to evaluate the impact of other parameterization on the results we can get from attacks. This is left to future works. All in all, evasion attacks showed interesting results and open new perspectives that we discuss in the Section 9.

We rely on centroids to deal with class imbalance (see Section 5.4). The centroid method has pros and cons: centroids are easy and quick to compute, new configurations tend to follow the same distribution as they result in more densely populated clusters and on rare occasions, make clusters expand a little bit. However, new configurations may not be realistic, since they do not provide so much diversity – centroids, by definition, lie in the middle of the cluster of points. Since our goal is only to limit imbalance in the available configurations, this technique is appropriate while maintaining the initial distribution of configurations. However, we are aware that other data augmentation techniques can be used.

The number of configurations to generate is also arguable. We tried to choose numbers for RQ1 that were large enough to be meaningful. RQ2 uses fewer adversarial configurations since, by design, the goal of evasion attacks is to produce configurations that do not follow the same distribution of data as the ones gathered by sampling configurations from a FM. Including too many adversarial configurations would make classification performance to drop drastically (being the goal of adversarial attacks). We chose 25 configurations for both our case studies according to the training set sizes and yet, we can already see an impact on the performances of the classifier.

The quality of the targeted ML classifier is a threat to our experiments. In particular, a too weak classifier would make numerous predictions errors and thus dramatically ease the task of putting it in default. On the other end, a too strong classifier (especially when the all the variants can be enumerated

in the case of JHipster) is very difficult to attack as exhaustive classification is possible. Many factors can influence an ML classifier: the underlying ML algorithm, the hyper-parameters or the training set used to train it. To mitigate this threat, we took many precautions for training an accurate ML classifier (see Section 5.4). As future work, we plan to investigate whether sampling strategies (*e.g.,* adversarial training, see Section 9.1, or t-wise, see Section 10) used to gather the training set can have impact on the quality of the ML classifier and in turn affect the effectiveness of adversarial configurations.

8.2 External threats

Compared to our previous work [85], we have added an extra case study to our experiments, it does not seem much but it gives insights on the generalization capability of the adaptation of adversarial attacks (and in particular evasion attacks) to the world of SPLs. Also, we chose a configurable system that was completely uncorrelated to the world of images and multimedia and which defines more cross-tree constraints than MOTIV, requiring additional effort to take them into account. Calls to SAT/SMT solvers are unpractical due to feature heterogeneity and the frequency of validity checks. Benchmarks of large and real-world feature models can be considered if we are only interested in sampling aspects [52,82].

We considered accuracy as our main performance measure. Accuracy is the standard measure used in the advML literature [7,61,17,19,18,41] to assess the impact of attacks.

SecML is a library gathering multiple implementations of adversarial attacks. In our experiments, we only focused on a single implementation which can already be parameterized in various different ways; other implementations also include parameters that need to be set. We decided to keep the implementation that seems the closest to the initial evasion algorithm (*i.e.,* CEvasionAttackPGD) but other provided implementations might be considered and their results compared to the ones reported in this work.

9 Discussions

Adversarial configurations pinpoint areas of the configuration space where the ML classifier fails or has low confidence in its prediction. We qualitatively discuss what *the existence of adversarial configurations suggests for an SPL* and how adversarial configurations can be practically used within the two case studies (MOTIV and JHipster).

9.1 Adversarial training

Developers of MOTIV and JHipster may legitimately want to improve their ML classifier by making it more robust to attacks (*i.e.,* adversarial configu-

rations). Previous work on advML [21,7,42,34,57] proposed different defense strategies in presence of adversarial configurations. Adversarial training is a specific category of defense: the training sample is augmented with adversarial examples to make ML models more robust. In our case study, it consists of applying our attack generator and re-inject adversarial configurations as part of the original training set.

We saw in *RQ2* that, when adversarial configurations are introduced in the training set, even moderately aggressive attacks may improve the ML classifier performance but also decrease it in some settings (notably with SecML on MOTIV). Indeed, we acknowledge that our *adversarial generators have simply not been designed for this defensive task and rather excel in triggering misclassifications*. It opens two perspectives. The first is to apply other, more effective defense mechanisms (such as manifold projections, stochasticity, or prepossessing [21,7,42,34,57]). The second and most promising perspective is to adapt adversarial ML knowledge with "friendly" rather than malicious attacks. That is, instead of only fooling the ML classifier, an additional objective would be to generate configurations as part of the training set that can improve its learning accuracy.

9.2 Improvement of the testing oracle

MOTIV. The labeling of videos as acceptable/non-acceptable is approximated by the ML classifier. If the testing oracle is not precise enough, it is likely that the approximation performs badly. In the MOTIV case, oracles are an approximation of the human perception system which in turn could be seen as an approximation of the real separation between acceptable images and non-acceptable ones regarding a specific task. Object recognition should potentially work on an infinite number of input images which makes the construction of a "traditional" oracle (a function that is able to give the nature of every single input) challenging. Testing oracles for an SPL are programs that may fail on some specific configurations. Adversarial configurations can lead to "cases" (videos) for which the oracle has not been designed and tested. Such configurations may thus provide insights to improve such oracles.

Specifically, MOTIV's developers can revise the visual assessment procedure to determine what a video of sufficient quality means [35,86]. Adversarial configurations can help in understanding the limits of the procedure over specific videos. The examples presented in Figure 5, page 14 are not resulting from attacks but were allowed by the MOTIV generator. In these examples, we were able to see that colors were not natural and thus need to be constrained. Another way to see it is that the testing oracle failed to detect these colors were unnatural. The oracle that was used in [86] focused only on avoiding too much blur or noise in images. It did not care about the realism of the colors, thus, to avoid these examples later, a new oracle could be developed to check automatically the range of values of features defining colors and integrated into the process. Based on the review of problematic cases, MOTIV's

developers can evolve the oracle or its parameters' value. The oracle relies on a Fourier transformation to assess the visual properties of an image. In the Fourier space, frequencies are represented as circles centered in the middle of the image. The oracle computes the distribution of frequencies from the center of the image (in the Fourier space) to the edges. Then this distribution is compared to a model of perfect homogeneous distribution (representing an ideal image). The allowed deviation from the ideal distribution is controlled by a threshold. Adversarial configurations exhibit important deviations from the ideal and can be exploited to set up this threshold more finely. An open problem is to find a way to control adversarial displacements such that we are able to ensure that the generated adversarial configurations are diverse enough to cover different visual cases. This level of control is left for future work.

JHipster. In a previous endeavor with JHipster, we learned that building a configuration-aware testing infrastructure requires a substantial effort in terms of human resources (8 man-months) [45]. Specifically, we spend significant time in engineering the right testing oracle to avoid false-positive failures *e.g.,* configurations that do not build because the testing environment has not been properly set up. Adversarial configurations can serve as new cases for investigating the quality of the testing environment. Overall, we recommend using adversarial configurations early in the quality assurance process of JHipster. It can help developers to debug and fix the testing environment by focusing on (potentially) problematic cases. This requires to model characteristics of the environment as part of the variability model.

9.3 Improvement of the variability model

While generating adversarial configurations, SPL practitioners can gain insights into whether the feature model is under or over-constrained. Looking at modified features of adversarial configurations (see *RQ2*), practitioners can observe that the same patterns arise involving some features or combinations of features. Such behavior typically indicates that constraints are missing – some configurations are allowed despite they should not be but it was never specifically defined as such in the variability model. Conversely, adversarial configurations can also help to identify which constraints can be relaxed. Some constraints may be an over-approximation of what was really expected. In this case, some configurations are wrongly forbidden as they would provide acceptable performances if they were tested. Evasion attacks may provide such configurations, starting from valid and acceptable configurations and moving towards a priori non-acceptable configurations. Because of the use of evasion attacks, resulting configurations may, in fact, be acceptable and valid and thus should not be forbidden in the variability model. A careful analysis of resulting configurations, by experts, is needed to ensure (or not) that they are really non-acceptable.

In the *MOTIV* case, one can envision to synthesize constraints and reinforce the variability model. Two strategies are conceivable for synthesizing

constraints: (1) out of adversarial configurations by mining (combinations of) features that are frequently present; (2) out of the new ML classifier, retrained with adversarial configurations, if the underlying ML model is interpretable. Right now, we cannot rely on decision trees since evasion attacks are harder to implement for such ML models (see Section 3). We leave as future work the investigation of the two strategies and current limitations.

In the *JHipster* case, the reverse-engineering feature model was validating that every allowed configuration was indeed supported by the JHipster configurator [45]. We therefore had an "exact" approach to create a correct variability model and advML is unhelpful. It is in contrast to the MOTIV case, mainly because the configuration space contains less valid configurations (only thousands of configurations compared to the 10^{314} configurations).

9.4 Improvement of the variability implementation

Features of *MOTIV* are implemented in Lua [46]. An incorrect implementation can be the cause of non-acceptable configurations either because of bugs in individual features or undesired feature interactions. In the case of MOTIV, even if some variability-related bugs may exist, especially generating video sequences out of scene content descriptions (*i.e.,* configurations), we rather considered that the cause of non-acceptable videos was due to the variability model and that the solution was to add constraints preventing them.

JHipster has a diverse stack of technologies and finding which of them are the source of bugs and issues is not trivial. Previous work demonstrated that it was possible with relatively few configurations (1% of the total number of configurations) to cover all interaction bugs. The role of adversarial configurations as a bug-finding sampling technique is worthy of future analyses. From this perspective, interesting future work is to compare adversarial configurations with sampling strategies developed in the SPL testing community (see Section 10).

9.5 AdvML and taking constraints of the variability model into account

The most significant part of our adaptation of adversarial techniques to SPL engineering resided in constraint handling. The lack of native support of advML algorithms may be traced to their original application domain: images. Data is homogeneous: dimensions are likely to be defined on the same range of values, all dimensions can be encoded similarly with values used for upper and lower bounds equal for all dimensions. Additionally, most constraints are "soft constraints": a white pixel on a black image does not make it invalid. In such a domain, the need for advanced constraint solvers does not exist.

In SPLs, constraints are key and we have to deal with heterogeneous features which in turn induces constraints during encoding. JHipster has only

Boolean features but a large number of dependencies expressed as cross-tree constraints while MOTIV comprises heterogeneous features but few constraints. We had to adopt different constraint handling strategies for each case study.

JHipster. We enforce constraints after each configuration displacement (intermediate point in the attack), that is resetting each feature value outside allowed ranges to a default acceptable one or letting the configuration unchanged if the configuration is valid. With a high number of constraints, this solution seemed wiser, as the risk of having to change all the values at the end of the attack is high, rendering the attack ineffective (since value resets to due constraints may hamper the relevance of the attack).

MOTIV. The low number of constraints in MOTIV allow for a post-attack strategy: the chances of having an invalid configuration are lower and it is also likely that only a few features are violating the constraints. The implications of changing these values to make them acceptable are limited regarding attack relevance.

While we believe that the above strategies were both simple and efficient, a more general and systematic approach relying on top of a SAT/SMT solver is to be devised. One can consider a variety of intermediate enforcement strategies such as implementing minimal configuration fixes [90].

9.6 Execution time

On one hand, our dedicated algorithm relies on a fixed number of computations. This allows for predictable yet non-optimal execution time. On the other hand, SecML optimizes its number of executions but may suffer from convergence issues. In the following, we discuss the parameters affecting the performance of both algorithms.

We measured some of the execution time (*i.e.,* when computations are minimal and when they are maximal). The main parameter to impact computation time in the dedicated algorithm is the number of displacements allowed (nb_disp). Setting to larger values will require extra computations to reach the final position, especially since we did not used any early stopping mechanism. Reported computation times range between 10 minutes when nb_disp is set to 20 to about $1h15$ when it is set to 100 with MOTIV. Even if nb_disp is not the only parameter impacting the effectiveness of the attacks, setting it to a larger value increases chances to generate configurations that misclassified which has to be confronted with the fact that computations are longer.

When using SecML, the η parameter as well as the d_max value can have big impacts on computation time. d_max sets the maximum amount of perturbations that can be added to the original configuration; therefore, in some sense, it could be seen as a combination of the nb_disp and the step size t of the dedicated implementation. If d_max is set to large values, it meets that the boundaries to displacements expand, potentially allowing for more iterations when using the gradient descent approach. On the other hand, η is

a parameter used in the gradient descent algorithm, it can be viewed as the resolution of a search grid on which the direction of the gradient is computed. Higher values will make the resolution coarse while lower values make the search more fine-grained. A fine-grained search will take more time, computing for more possibilities at the cost of potentially getting stuck in sub-optimal spaces while a coarse resolution will be quicker to compute but might miss the optimal solution. Therefore, there is a trade-off to find. We set the value of η to 0.01 based on the nature and number of features representing our data, it gave sufficiently good results but it might not the best value. Anyhow, we observed that η can have a big impact on computation cost. With a default value of 0.001, our computations could last up to 50 minutes while with the chosen value of 0.01, computation times can be boiled down to about 10 minutes. The shortest execution time that we have observed was under a minute when η was set to 0.01 and d_max set to 0.1 with MOTIV.

Finding the optimal values to reach results of good quality at a reasonable computation cost is a challenge: for SPL practitioners that do not know very well how advML works or have little experience in tuning their advML technique.

Finally, reported times can be reduced by reducing the number of configurations to generate. In our experiments, they were set to the same amount regardless of the value assigned with these parameters, we only changed this number per case study. Computation times were significantly lower when executing attacks on JHipster (aiming to generate 1000 configurations) w.r.t. MOTIV (needed 4000 configurations).

9.7 White-box or black-box

While the "original" adversarial setting was designed to support malicious scenarios (*e.g.,* hackers that would try to outperform the system under attack) leading to attack models, here we proposed a defensive method that should be used along with the developer of the SPL system in order to improve knowledge about the SPL but also to improve guidance for users trying to find a suitable configuration for their task-at-hand. That is, the term "adversarial" may be disturbing at first glance since our context is different from an intrusion detection system that would be attacked by maleficent users. Our method can leverage some access to typical parts of a SPL such as the configurator or the underlying variability model. This knowledge of the variability model is necessary to generate configurations that are valid and recognized by the classifier and variant generator. In the case where a variability model would not be available, it is still possible to reverse-engineer it [79, 78, 56, 1, 9]. After that, it is all about configurations. The system can be viewed as a distant system on which query can be performed about the acceptability of a configuration. Once the query is launched with the configuration, the system tries to generate it, runs some tests to decide whether it is acceptable and/or asks the trained classifier to predict its acceptability. Adversarial attacks will leverage

information given by the answer to the query to build new configurations that try to evade. In this sense, we do not see this method as white-box but mostly black-box, opening a broader use.

10 Related Work

Our contribution is at the crossroad of (adversarial) ML, constraint mining, variability modeling, and testing.

Testing and learning SPLs. Testing all configurations of an SPL is most of time challenging and sometimes impossible, due to the exponential number of configurations [87,59,55,12,89,44,69,25,3]. ML techniques have been developed to reduce cost, time and energy of deriving and testing new configurations using inference mechanisms. For instance, regression models can be used to perform performance prediction of configurations that have not been generated yet [80,74,43,11,81,62,68]. In our work, we consider qualitative properties (*e.g.,* the presence of bugs in a configuration), address a statistical classification problem, and target classifiers of an SPL.

Siegmund *et al.* [82] reviewed ML approaches on variability models. They propose THOR, a tool for synthesizing realistic attributed variability models. An important issue in this line of research is to assess the robustness of ML on variability models. Our work specifically aims to improve ML classifiers of SPL. None of these bodies of work use adversarial ML neither the possible impact that adversarial configurations could have on the predictions. Besides, there are many machine learning-based approaches that seek to mine constraints with ML [91,53,40,68]. In [86,84,2,6], we rely on supervised ML to discover and retrieve constraints that were not originally expressed before in a variability model. We typically used decision trees to create a boundary between the configurations that should be discarded and the ones that are allowed. Our work aims to improve such ML-based approaches through the generation of adversarial configurations.

Sampling configurations. Several sampling strategies have been proposed in the literature about SPLs and configurable systems [87,5,89]. Some works consider sampling with the specific goal of testing configurations of an SPL. There is no learning phase and the objective is mostly to find and cover as many configuration faults (bugs) as possible. For instance, Medeiros *et al.* compared 10 sampling algorithms to detect bugs in configurable systems [59]. Varshosaz *et al.* [89] conducted a survey of sampling for testing configurable systems. Pereira *et al.* [5] and Kaltenecker *et al.* [49] review several sampling strategies in the context of learning performance models of configuration spaces. Random sampling is used to cover the configuration space *uniformly.* Oh *et al.* [63] rely on binary decision diagrams to compactly represent a configuration space but satisfiability (SAT) solvers can also be used. For instance, UniGen [26] uses hashing-based functions to synthesize configurations' samples in a nearly uniform manner with strong theoretical guarantees. Plazar *et al.* [70] showed that state-of-the-art algorithms are either not able to produce

any sample or unable to generate uniform samples for the SAT instances considered. When random sampling is not applicable, several alternate techniques have been proposed typically by sacrificing some uniformity for a substantial increase in performance. Coverage-based sampling aims to optimize the sample with regards to a coverage criterion. Many criteria can be considered such as statement coverage that requires the analysis of the source code. The t-wise sampling [48,30] strategy selects configurations to cover all combinations of t selected options. For instance, pair-wise (t=2) sampling covers all pairwise combinations of options being selected. There are different methods to compute t-wise sampling. Kaltenecker *et al.* [49] propose distance-based sampling and diversified distance-based sampling. The idea is to cover the configuration space by selecting configurations according to a given probability distribution (typically a uniform distribution) and a distance metric.

Generating adversarial configurations can be seen as a way to sample the configuration space. A major difference is that adversarial techniques are specifically designed and crafted to put in trouble an ML classifier, by exploiting its intrinsic properties and possible lacks. We have shown that such configurations can even improve an ML classifier and thus avoid non-acceptable products of an SPL. It is worth noticing that adversarial configurations can be symbolically exploited, *i.e.,*, without having to derive, test, or measure the additional corresponding products of an SPL.

Adversarial ML can be seen as a set of security assessment and reinforcement techniques helping to better understand flaws and weaknesses of ML algorithms. Typical scenarios which use adversarial learning are: network traffic monitoring, spam filtering, malware detection [7,17,19,18,16,20] and more recently autonomous cars and object recognition [92,66,36,64,77,54,37]. In such works, authors suppose that a system uses ML in order to perform a classification task (*e.g.,* differentiate emails as spams and non-spams) and some malicious people try to fool such a classification system. These attackers can have knowledge on the system such as the dataset used, the kind of ML technique that is used, or the description of data among others. The attack then consists of crafting a data point in the description space that the ML algorithm will misclassify. Recent works [41] used adversarial techniques to strengthen the classifier by specifically creating data that would induce such kind of misclassification. In this article, we propose to use a similar approach but adapted to SPL engineering: adversarial techniques can be used to strengthen the SPL (including ML classifier, variability model, implementation, and testing oracle over products). In addition, to broaden the applicability of advML, we have investigated the specific problem of generating examples (configurations) that should be both adversarial and conform to the logical constraints of an SPL. Some approaches such as DeepXplore [67] or DeepTest [88] rely on generative adversarial networks (GAN) to synthesize new test cases (images) based on the notion of neuron coverage. This line of work does not specifically generate adversarial configurations and does not aim to support the engineering of configurable systems or SPLs.

To summarize and to the best of our knowledge: on the one hand, numerous techniques have been developed to test or learn software configuration spaces, but none of them consider advML. On the other hand, the application and assessment of advML for engineering SPLs and configurable software systems have not caught specific attention.

11 Conclusion

Machine learning (ML) techniques are increasingly used in software product line (SPL) engineering, to cope with the tremendous number of possible configurations that SPL can handle, or for their ability to predict whether a configuration (and its associated program variant) meets quality requirements.

Yet, ML techniques can make prediction errors in areas where the confidence in the classification is low. Adversarial techniques take advantage of this to create data points that force classification errors.

Our goal was to investigate the relevance of adversarial techniques in an SPL context. We conducted an empirical assessment on two very different case studies: a video generator (MOTIV) and full-stack web application configurator and generator (JHipster).

A first lesson learned is that current adversarial techniques require adaptation to be usable for SPLs, notably regarding constraints expressed in the feature models. Enforcing such constraints is key to prevent the generation of invalid adversarial configurations. We designed ad-hoc solutions to enforce SPL constraints for our case studies.

The second lesson learned is that adversarial techniques are indeed relevant for SPLs, hereby fulfilling our goal. We demonstrated that our adapted algorithms lead to valid and misclassified configurations. Additionally, when included in the training set, few adversarial configurations were sufficient to affect the prediction performance of the classifier, positively in some attack settings.

This empirical study is the first step towards a quality assurance and repair framework for ML-enabled SPLs relying on adversarial machine learning. To achieve this objective, we envision the following research directions:

1. Investigate how constraint support can be systematized, notably by integrating constraint solvers in SecML;
2. Explore other kinds of adversarial attacks and design them so that they either have a positive or negative impact on the classifier's accuracy when included in the training set;
3. Sampling configurations is already an important step when testing SPLs and training a ML model, it remains equally important in our framework. The impact of different sampling strategies should be assessed as a parameter of this method with regards to the efficiency to produce adversarial configurations that are misclassified. A baseline for this comparison could be the results we showed in this paper.

4. Finally, consider adversarial configuration generation as a sampling technique and compare it to well-known sampling techniques (*e.g.,* random, t-wise).

Acknowledgements Gilles Perrouin is an FNRS Research Associate. This research was partly supported by EOS Verilearn project grant no. O05518F-RG03. This research was also funded by the ANR-17-CE25-0010-01 VaryVary project.

References

1. Acher, M., Cleve, A., Perrouin, G., Heymans, P., Vanbeneden, C., Collet, P., Lahire, P.: On extracting feature models from product descriptions. In: Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, VaMoS '12, pp. 45–54. ACM, New York, NY, USA (2012). DOI 10.1145/2110147.2110153. URL http://doi.acm.org/10.1145/2110147.2110153
2. Acher, M., Temple, P., Jezequel, J.M., Galindo, J.A., Martinez, J., Ziadi, T.: Varylatex: Learning paper variants that meet constraints. In: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems, pp. 83–88. ACM (2018)
3. Al-Hajjaji, M., Benduhn, F., Thüm, T., Leich, T., Saake, G.: Mutation operators for preprocessor-based variability. In: Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems, Salvador, Brazil, January 27 - 29, 2016, pp. 81–88 (2016). DOI 10.1145/2866614.2866626. URL https://doi.org/10.1145/2866614.2866626
4. Alférez, M., Acher, M., Galindo, J.A., Baudry, B., Benavides, D.: Modeling variability in the video domain: language and experience report. Softw. Qual. J. **27**(1), 307–347 (2019). DOI 10.1007/s11219-017-9400-8. URL https://doi.org/10.1007/s11219-017-9400-8
5. Alves Pereira, J., Acher, M., Martin, H., Jézéquel, J.M.: Sampling Effect on Performance Prediction of Configurable Systems: A Case Study. In: 11th International Conference on Performance Engineering (ICPE'20) (2020). URL https://hal.inria.fr/hal-02356290
6. Amand, B., Cordy, M., Heymans, P., Acher, M., Temple, P., Jézéquel, J.M.: Towards learning-aided configuration in 3d printing: Feasibility study and application to defect prediction. In: Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems, p. 7. ACM (2019)
7. Barreno, M., Nelson, B., Sears, R., Joseph, A.D., Tygar, J.D.: Can machine learning be secure? In: Proceedings of the 2006 ACM Symposium on Information, computer and communications security, pp. 16–25. ACM, New York, NY, USA (2006)
8. Batory, D.S.: Feature models, grammars, and propositional formulas. In: SPLC'05, *LNCS*, vol. 3714, pp. 7–20. Springer, Berlin, Germany (2005)
9. Bécan, G., Behjati, R., Gotlieb, A., Acher, M.: Synthesis of attributed feature models from product descriptions. In: SPLC'15 (2015)
10. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. J. Log. Algebr. Meth. Program. **85**(2), 287–315 (2016). DOI 10.1016/j.jlamp.2015.11.006. URL https://doi.org/10.1016/j.jlamp.2015.11.006
11. ter Beek, M.H., Fantechi, A., Gnesi, S., Semini, L.: Variability-based design of services for smart transportation systems. In: Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II, pp. 465–481 (2016). DOI 10.1007/978-3-319-47169-3_38. URL https://doi.org/10.1007/978-3-319-47169-3_38

12. ter Beek, M.H., Legay, A.: Quantitative variability modeling and analysis. In: Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS '19, pp. 13:1–13:2. ACM, New York, NY, USA (2019). DOI 10.1145/3302333.3302349. URL http://doi.acm.org/10.1145/3302333.3302349
13. Bellman, R.: Dynamic Programming, 1 edn. Princeton University Press, Princeton, NJ, USA (1957)
14. Benavides, D., Segura, S., Ruiz-Cortes, A.: Automated analysis of feature models 20 years later: a literature review. Information Systems **35**(6), 615–636 (2010)
15. Berger, T., Rublack, R., Nair, D., Atlee, J.M., Becker, M., Czarnecki, K., Wąsowski, A.: A survey of variability modeling in industrial practice. In: Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13, pp. 7:1–7:8. ACM, New York, NY, USA (2013). DOI 10.1145/2430502.2430513. URL http://doi.acm.org/10.1145/2430502.2430513
16. Biggio, B., Corona, I., Maiorca, D., Nelson, B., Šrndić, N., Laskov, P., Giacinto, G., Roli, F.: Evasion attacks against machine learning at test time. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, pp. 387–402. Springer Berlin, Berlin, Heidelberg (2013)
17. Biggio, B., Didaci, L., Fumera, G., Roli, F.: Poisoning attacks to compromise face templates. In: 2013 International Conference on Biometrics (ICB), pp. 1–7. IEEE, New York, USA (2013). DOI 10.1109/ICB.2013.6613006
18. Biggio, B., Fumera, G., Roli, F.: Pattern recognition systems under attack: Design issues and research challenges. International Journal of Pattern Recognition and Artificial Intelligence **28**(07), 1460002 (2014)
19. Biggio, B., Fumera, G., Roli, F.: Security evaluation of pattern classifiers under attack. IEEE transactions on knowledge and data engineering **26**(4), 984–996 (2014)
20. Biggio, B., Nelson, B., Laskov, P.: Poisoning attacks against support vector machines. In: Proceedings of the 29th International Coference on International Conference on Machine Learning, ICML'12, pp. 1467–1474. Omnipress, USA (2012). URL http://dl.acm.org/citation.cfm?id=3042573.3042761
21. Biggio, B., Roli, F.: Wild patterns: Ten years after the rise of adversarial machine learning. Pattern Recognition **84**, 317–331 (2018)
22. Bodden, E., Tolêdo, T., Ribeiro, M., Brabrand, C., Borba, P., Mezini, M.: SPL$^{\text{lift}}$: statically analyzing software product lines in minutes instead of years. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013, pp. 355–364. ACM, New York, USA (2013). DOI 10.1145/2491956.2491976. URL http://doi.acm.org/10.1145/2491956.2491976
23. Boucher, Q., Classen, A., Faber, P., Heymans, P.: Introducing tvl, a text-based feature modelling. In: D. Benavides, D.S. Batory, P. Grünbacher (eds.) Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria, January 27-29, 2010. Proceedings, *ICB-Research Report*, vol. 37, pp. 159–162. Universität Duisburg-Essen, Essen, Germany (2010). URL http://www.vamos-workshop.net/proceedings/VaMoS_2010_Proceedings.pdf
24. Brown, T., Mane, D., Roy, A., Abadi, M., Gilmer, J.: Adversarial patch. https://arxiv.org/pdf/1712.09665.pdf (2017)
25. Carvalho, L., Guimarães, M.A., Ribeiro, M., Fernandes, L., Al-Hajjaji, M., Gheyi, R., Thüm, T.: Equivalent mutants in configurable systems: An empirical study. In: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS 2018, Madrid, Spain, February 7-9, 2018, pp. 11–18 (2018). DOI 10.1145/3168365.3168379. URL https://doi.org/10.1145/3168365.3168379
26. 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)
27. Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: Smote: synthetic minority over-sampling technique. Journal of artificial intelligence research **16**, 321–357 (2002)
28. Classen, A., Boucher, Q., Heymans, P.: A text-based approach to feature modelling: Syntax and semantics of TVL. Science of Computer Programming, Special Issue on Software Evolution, Adaptability and Variability **76**(12), 1130–1143 (2011)

29. Clements, P., Northrop, L.M.: Software Product Lines : Practices and Patterns. Addison-Wesley Professional, Boston, USA (2001)
30. Cohen, M.B., Dwyer, M.B., Society, I.C.: Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints : A Greedy Approach. IEEE Transactions on Software Engineering (2008)
31. Davril, J.M., Heymans, P., Bécan, G., Acher, M.: On Breaking The Curse of Dimensionality in Reverse Engineering Feature Models. In: 17th International Configuration Workshop, *17th International Configuration Workshop*, vol. 17th International Configuration Workshop. Vienna, Austria (2015). URL `https://hal.inria.fr/hal-01243571`
32. Demontis, A., Melis, M., Pintor, M., Jagielski, M., Biggio, B., Oprea, A., Nita-Rotaru, C., Roli, F.: On the intriguing connections of regularization, input gradients and transferability of evasion and poisoning attacks. CoRR **abs/1809.02861** (2018). URL `http://arxiv.org/abs/1809.02861`
33. Demontis, A., Melis, M., Pintor, M., Jagielski, M., Biggio, B., Oprea, A., Nita-Rotaru, C., Roli, F.: Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In: 28th USENIX Security Symposium (USENIX Security 19). USENIX Association, Santa Clara, CA (2019). URL `https://www.usenix.org/conference/usenixsecurity19/presentation/demontis`
34. Dhillon, G.S., Azizzadenesheli, K., Lipton, Z.C., Bernstein, J., Kossaifi, J., Khanna, A., Anandkumar, A.: Stochastic activation pruning for robust adversarial defense. arXiv preprint arXiv:1803.01442 (2018)
35. Dosselman, R.W., Yang, X.D.: No-reference noise and blur detection via the fourier transform. Tech. rep., University of Regina, CANADA (2012)
36. Elsayed, G.F., Shankar, S., Cheung, B., Papernot, N., Kurakin, A., Goodfellow, I., Sohl-Dickstein, J.: Adversarial examples that fool both human and computer vision. arXiv preprint arXiv:1802.08195 (2018)
37. Evtimov, I., Eykholt, K., Fernandes, E., Kohno, T., Li, B., Prakash, A., Rahmati, A., Song, D.: Robust physical-world attacks on deep learning models. arXiv preprint arXiv:1707.08945 **1** (2017)
38. Galindo, J.A., Alférez, M., Acher, M., Baudry, B., Benavides, D.: A variability-based testing approach for synthesizing video sequences. In: International Symposium on Software Testing and Analysis, ISSTA 2014, pp. 293–303. ACM (2014). DOI 10.1145/2610384.2610411. URL `http://doi.acm.org/10.1145/2610384.2610411`
39. Galindo Duarte, J.A., Alférez, M., Acher, M., Baudry, B., Benavides, D.: A Variability-Based Testing Approach for Synthesizing Video Sequences. In: ISSTA '14: International Symposium on Software Testing and Analysis. San José, California, United States (2014). URL `https://hal.inria.fr/hal-01003148`
40. Gargantini, A., Petke, J., Radavelli, M.: Combinatorial interaction testing for automated constraint repair. In: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 239–248. IEEE (2017)
41. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial nets. In: Advances in neural information processing systems, pp. 2672–2680 (2014)
42. Guo, C., Rana, M., Cisse, M., van der Maaten, L.: Countering adversarial images using input transformations. arXiv preprint arXiv:1711.00117 (2017)
43. Guo, J., Czarnecki, K., Apel, S., Siegmund, N., Wasowski, A.: Variability-aware performance prediction: A statistical learning approach. In: ASE (2013)
44. 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 10.07980. URL `https://hal.inria.fr/hal-01829928`. Empirical Software Engineering journal
45. 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 **24**(2), 674–717 (2019). DOI 10.1007/s10664-018-9635-4. URL `https://doi.org/10.1007/s10664-018-9635-4`
46. Ierusalimschy, R.: Programming in Lua, Second Edition. Lua.Org (2006)
47. JHipsterTeam: Jhipster website (2020). URL `https://jhipster.github.io`. Https://jhipster.github.io, accessed Jan. 2020.

48. Johansen, M.F., Haugen, O.y., Fleurey, F.: An algorithm for generating t-wise covering arrays from large feature models. SPLC'12 (2012)
49. 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)
50. Kaner, C., Bach, J., Pettichord, B.: Lessons Learned in Software Testing. John Wiley & Sons, Inc., New York, NY, USA (2001)
51. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. rep., DTIC Document (1990)
52. Knüppel, A., Thüm, T., Mennicke, S., Meinicke, J., Schaefer, I.: Is there a mismatch between real-world feature models and product-line research? In: M. Tichy, E. Bodden, M. Kuhrmann, S. Wagner, J. Steghöfer (eds.) Software Engineering und Software Management 2018, Fachtagung des GI-Fachbereichs Softwaretechnik, SE 2018, 5.-9. März 2018, Ulm, Germany., *LNI*, vol. P-279, pp. 53–54. Gesellschaft für Informatik (2018). URL `https://dl.gi.de/20.500.12116/16312`
53. Krismayer, T., Rabiser, R., Grünbacher, P.: Mining constraints for event-based monitoring in systems of systems. In: ASE, pp. 826–831. IEEE Press (2017)
54. Kurakin, A., Goodfellow, I., Bengio, S.: Adversarial examples in the physical world. arXiv preprint arXiv:1607.02533 (2016)
55. Legay, A., Perrouin, G.: On quantitative requirements for product lines. In: Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems, VAMOS '17, pp. 2–4. ACM, New York, NY, USA (2017). DOI 10.1145/3023956.3023970. URL `http://doi.acm.org/10.1145/3023956.3023970`
56. Lopez-Herrejon, R.E., Galindo, J.A., Benavides, D., Segura, S., Egyed, A.: Reverse engineering feature models with evolutionary algorithms: An exploratory study. In: SSBSE'12, *LNCS*, vol. 7515, pp. 168–182. Springer (2012)
57. Madry, A., Makelov, A., Schmidt, L., Tsipras, D., Vladu, A.: Towards deep learning models resistant to adversarial attacks. arXiv preprint arXiv:1706.06083 (2017)
58. Mann, H.B., Whitney, D.R.: On a test of whether one of two random variables is stochastically larger than the other. The annals of mathematical statistics pp. 50–60 (1947)
59. 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, New York, NY, USA (2016). DOI 10.1145/2884781.2884793. URL `http://doi.acm.org/10.1145/2884781.2884793`
60. Nadi, S., Berger, T., Kästner, C., Czarnecki, K.: Mining configuration constraints: static analyses and empirical results. In: 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014, pp. 140–151 (2014). DOI 10.1145/2568225.2568283. URL `http://doi.acm.org/10.1145/2568225.2568283`
61. Nelson, B., Barreno, M., Chi, F.J., Joseph, A.D., Rubinstein, B.I., Saini, U., Sutton, C.A., Tygar, J.D., Xia, K.: Exploiting machine learning to subvert your spam filter. LEET **8**, 1–9 (2008)
62. Oh, J., Batory, D.S., Myers, M., Siegmund, N.: Finding near-optimal configurations in product lines by random sampling. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017, pp. 61–71 (2017). DOI 10.1145/3106237.3106273. URL `http://doi.acm.org/10.1145/3106237.3106273`
63. Oh, J., Batory, D.S., Myers, M., Siegmund, N.: Finding near-optimal configurations in product lines by random sampling. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017, pp. 61–71 (2017)
64. Papernot, N., McDaniel, P., Jha, S., Fredrikson, M., Celik, Z.B., Swami, A.: The limitations of deep learning in adversarial settings. In: 2016 IEEE European Symposium on Security and Privacy (EuroS P), pp. 372–387 (2016). DOI 10.1109/EuroSP.2016.36
65. Pascual, G.G., Lopez-Herrejon, R.E., Pinto, M., Fuentes, L., Egyed, A.: Applying multiobjective evolutionary algorithms to dynamic software product lines for reconfiguring mobile applications. Journal of Systems and Software **103**, 392–411 (2015)

66. Pei, K., Cao, Y., Yang, J., Jana, S.: Deepxplore: Automated whitebox testing of deep learning systems. In: Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, pp. 1–18. ACM, New York, NY, USA (2017). DOI 10.1145/3132747.3132785. URL http://doi.acm.org/10.1145/3132747.3132785
67. Pei, K., Cao, Y., Yang, J., Jana, S.: Deepxplore: Automated whitebox testing of deep learning systems. In: Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, pp. 1–18. ACM, New York, NY, USA (2017). DOI 10.1145/3132747.3132785. URL http://doi.acm.org/10.1145/3132747.3132785
68. Pereira, J.A., Martin, H., Acher, M., JÃĺzÃĺquel, J.M., Botterweck, G., Ventresque, A.: Learning software configuration spaces: A systematic literature review (2019)
69. Plazar, Q., Acher, M., Perrouin, G., Devroey, X., Cordy, M.: Uniform sampling of SAT solutions for configurable systems: Are we there yet? In: 12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019, pp. 240–251 (2019). DOI 10.1109/ICST.2019.00032. URL https://doi.org/10.1109/ICST.2019.00032
70. 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
71. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag (2005)
72. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2020). URL https://www.R-project.org/
73. Raible, M.: The JHipster mini-book. C4Media (2015)
74. Sarkar, A., Guo, J., Siegmund, N., Apel, S., Czarnecki, K.: Cost-efficient sampling for performance prediction of configurable systems (t). In: ASE'15 (2015)
75. Schobbens, P.Y., Heymans, P., Trigaux, J.C.: Feature diagrams: A survey and a formal semantics. In: RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06), pp. 136–145. IEEE Computer Society, Washington, DC, USA (2006). DOI http://dx.doi.org/10.1109/RE.2006.23
76. Schobbens, P.Y., Heymans, P., Trigaux, J.C., Bontemps, Y.: Generic semantics of feature diagrams. Comput. Netw. **51**(2), 456–479 (2007)
77. Sharif, M., Bhagavatula, S., Bauer, L., Reiter, M.K.: Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1528–1540. ACM (2016)
78. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Reverse engineering feature models. In: ICSE, pp. 461–470 (2011)
79. She, S., Ryssel, U., Andersen, N., Wasowski, A., Czarnecki, K.: Efficient synthesis of feature models. Information and Software Technology **56**(9) (2014)
80. Siegmund, N., Grebhahn, A., Kästner, C., Apel, S.: Performance-influence models for highly configurable systems. In: ESEC/FSE'15 (2015)
81. 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. (2013)
82. 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, ESEC/FSE 2017, pp. 268–278. ACM, New York, NY, USA (2017). DOI 10.1145/3106237.3106251. URL http://doi.acm.org/10.1145/3106237.3106251
83. Strüber, D., Rubin, J., Arendt, T., Chechik, M., Taentzer, G., Plöger, J.: Variability-based model transformation: formal foundation and application. Formal Asp. Comput. **30**(1), 133–162 (2018). DOI 10.1007/s00165-017-0441-3. URL https://doi.org/10.1007/s00165-017-0441-3
84. Temple, P., Acher, M., Jézéquel, J., Barais, O.: Learning contextual-variability models. IEEE Software **34**(6), 64–70 (2017). DOI 10.1109/MS.2017.4121211. URL https://doi.org/10.1109/MS.2017.4121211
85. Temple, P., Acher, M., Perrouin, G., Biggio, B., Jezequel, J.M., Roli, F.: Towards quality assurance of software product lines with adversarial configurations. In: Proceedings of

the 23rd International Systems and Software Product Line Conference - Volume A, SPLC '19, pp. 277–288. ACM, New York, NY, USA (2019). DOI 10.1145/3336294.3336309. URL http://doi.acm.org/10.1145/3336294.3336309

86. Temple, P., Galindo Duarte, J.A., Acher, M., Jézéquel, J.M.: Using Machine Learning to Infer Constraints for Product Lines. In: Software Product Line Conference (SPLC). Beijing, China (2016). DOI 10.1145/2934466.2934472. URL https://hal.inria.fr/hal-01323446
87. 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)
88. Tian, Y., Pei, K., Jana, S., Ray, B.: Deeptest: automated testing of deep-neural-network-driven autonomous cars. In: Proceedings of the 40th International Conference on Software Engineering, ICSE, pp. 303–314 (2018). DOI 10.1145/3180155.3180220. URL https://doi.org/10.1145/3180155.3180220
89. Varshosaz, M., Al-Hajjaji, M., Thüm, T., Runge, T., Mousavi, M.R., Schaefer, I.: A classification of product sampling for software product lines. In: Proceeedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018, pp. 1–13 (2018). DOI 10.1145/3233027.3233035. URL https://doi.org/10.1145/3233027.3233035
90. Xiong, Y., Hubaux, A., She, S., Czarnecki, K.: Generating range fixes for software configuration. In: 34th International Conference on Software Engineering (2012)
91. 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)
92. Zhang, M., Zhang, Y., Zhang, L., Liu, C., Khurshid, S.: Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, pp. 132–142. ACM, New York, NY, USA (2018). DOI 10.1145/3238147.3238187. URL http://doi.acm.org/10.1145/3238147.3238187