ML + FV = ♡?
A Survey on the Application of Machine Learning to Formal Verification

Moussa Amrani
University of Namur, Faculty of Computer Science, PReCiSE / NaDI
Rue Grangagnage, 21
Namur, Belgium 5000
Moussa.Amrani@unamur.be

Levi Lúcio
fortiss GmbH
Guerickestraße 25
München, Germany 80805
lucio@fortiss.org

Adrien Bibal
University of Namur, Faculty of Computer Science, PReCiSE / NaDI
Rue Grangagnage, 21
Namur, Belgium 5000
Adrien.Bibal@unamur.be

ABSTRACT

Formal Verification (Fv) and Machine Learning (Ml) can seem incompatible due to their opposite mathematical foundations and their use in real-life problems: Fv mostly relies on discrete mathematics and aims at ensuring correctness; Ml often relies on probabilistic models and consists of learning patterns from training data. In this paper, we postulate that they are complementary in practice, and explore how Ml helps Fv in its classical approaches: static analysis, model-checking, theorem-proving, and Sat solving. We draw a landscape of the current practice and catalog some of the most prominent uses of Ml inside Fv tools, thus offering a new perspective on Fv techniques that can help researchers and practitioners to better locate the possible synergies. We discuss lessons learned from our work, point to possible improvements and offer visions for the future of the domain in the light of the science of software and systems modeling.

CCS CONCEPTS

•General and reference → Validation; Verification; •Computing methodologies → Machine learning;

KEYWORDS

Formal Verification, Machine Learning

1 INTRODUCTION

Formal Verification (Fv) aims at guaranteeing correctness properties of software and hardware systems. In that sense, a system is safe with respect to the checked properties. Machine Learning (Ml) aims at learning patterns from training data for various purposes; the derived model generalizes from the data it was trained on. Both Fv and Ml are grounded on solid mathematical foundations: the former uses mostly discrete mathematics, fixpoints and abstractions to specify (concrete/abstract) semantics, properties of interest and the checking process itself; the latter uses in general continuous mathematics and/or probability theory to infer models. While they seem at first sight not suitable for each other, their relative and apparent opposition provides, just like in real life, the spark for a strong love story. This paper focuses on one part of the love story: what Ml brings to Fv to make it flourish, become more efficient and accurate, and face real-life challenges? While we are aware that the topic is broad and the ways in which Ml can help Fv are necessarily disparate, Fv newcomers and practitioners have currently no pointers to introduce them to the topic.

This paper is an attempt to provide a comprehensive survey of the various ways Ml contributes to enhance Fv tools' efficiency. To achieve this goal, we propose to catalog the challenges Fv faces that may be handled through Ml techniques, called *themes*, and characterize each theme with a corresponding Ml *task*, i.e. Ml problem categories (such as classification, regression, clustering, etc.), pointing for each theme to the relevant literature. To the best of our knowledge, no contributions in the literature currently exist that spans all the spectrum of the main Fv approaches (namely, Model-Checking, Theorem-Proving, Static Analysis, and to a certain extend, Sat-solving). By covering various approaches, we aim at extracting valuable, transversal lessons about general trends of Ml usage within Fv, as well as provide a high-level snapshot of the current practice in each Fv approach.

The main contributions of this paper are the following:

- We provide a catalog of themes for each Fv approach, presented in a systematic way: each theme details the corresponding Ml task, and provides a commented list of relevant contributions. An overview is available in Table 2.
- We analyze the literature to extract general observations on the use of Ml inside Fv tools, and to identify some trends and lessons, with an insight on what the future may be.
- We build a comprehensive and searchable repository of contributions found in the literature that can help the Fv community build a multi-level understanding of Ml usage in Fv tools. The repository is sorted according to various criteria (publication date, themes, and Ml tasks).

This paper is organized as follows. Section 3 describes the paper selection protocol, formulates research questions and discusses threats to validity. Section 4 analyses in detail the contributions we retrieved. Section 5 discusses how our findings answer the research questions before concluding in Section 7.

2 BACKGROUND

This section provides a high-level introduction to both Formal Verification (Fv) and Machine Learning (Ml), as the key actors in our survey. For further details, reference pointers are provided in each section.

2.1 Formal Verification (FV)

In its most classical form, Fv attempts to answer the following question: does a *behavioral model*, which reflects the evolution of the various variables of a system, satisfy a *specification* of a program, which consists in properties of interest characterizing error/undesired states.

Computing the system's so-called *concrete semantics* explicitly is in the general case impossible, since it is infinite even for very simple programs. Rather, Fv proceeds by *abstraction*, or *overapproximation* [28]: demonstrating that an abstraction never reaches forbidden values proves the fact that the actual executions are correct. However, *false alarms* (or *false positives*) may arise, i.e. errors due to an abstraction that is too coarse and that does not correspond to any actual system execution. Aside from extracting the behavioral model itself, one of the main difficulties in Fv is building abstractions that are sufficiently precise to avoid false alarms, but sufficiently simple to be automatically computed. This abstraction can take on many forms, leading to a variety of Fv approaches. In this paper we will concentrate on Static Analysis (Sa), Model Checking (Mc) and Theorem Proving (Tp). We also consider Sat solving (Sat): many Fv problems can be reduced to the satisfaction of Sat formulæ [77].

2.2 Machine Learning (ML)

Humans learn from experience. Car drivers learn by following instructions from driving school monitors, parents or friends, but also by identifying good behaviors in other drivers. Players learn chess or basket-ball by studying "good" and "bad" games, practicing the fundamental moves again and again, and by identifying best practices that ensure victory. Humans seem also naturally designed to extract patterns and features in what surrounds them. For instance, medical doctors provide diagnostics based on many anatomic and physiological variables such as body temperature or blood pressure – formally called *features* in Ml– available in patient data, trying to minimize death risk. Ml can be seen as a systematic way of solving a problem by optimizing some objective function using training data [67].

In this paper, we only consider three task categories in Ml: *supervised* and *unsupervised learning*, as well as *reinforcement learning*. Other categories (like *semi-supervised learning*), as well as many other Ml tasks in supervised/unsupervised learning exist, but the ones presented here cover all the themes we encountered while analyzing contributions of Ml in Fv approaches.

In *supervised learning*, a learning algorithm (*learner*) builds a predictive model during a training phase, based on features found in the training data, by optimizing a defined objective function. The learned model is then used in a subsequent phase as a predictor for new, previously unseen data. Tasks can be further classified into problems according to the nature of the predicted variable (also called *target*): categorical or continuous. For instance, in the medical domain, determining whether an Mri evidences a cancer is a *classification* task, because the answer is categorical (a boolean yes/no answer, but more elaborate classes may be possible); whereas determining which quantity of insulin should be injected into a patient's blood stream is a *regression* task, since the predicted variable is continuous.

In supervised learning, the target is known *a priori* and the learner minimizes some type of distance between the target and its prediction. On the contrary, in *unsupervised learning*, no target is given *a priori*: the Ml algorithm tries to find recurring patterns inside the data and the final quality judgment is ultimately human. The well-known *clustering* Ml task consists for instance in grouping elements in the dataset, but whether finding three clusters is better than finding five depends on the domain problem and cannot be answered fully automatically outside the context of the algorithm's use. Another task of this kind, the *item set finding* task, consists in finding items that may often co-occur together (e.g., buying butter and jam may often occur together with buying bread).

Finally, the third considered category is *reinforcement learning*, which "is learning what to do — how to map situations to actions — so as to maximize a numerical reward signal. The learner is not told which actions to take, as in many forms of machine learning, but instead must discover which actions yield the most reward by trying them" [92]. For instance, an AI agent taking the role of a human player in a computer game can learn how to complete a level by finding the action patterns leading to a minimization of penalties. These penalties may be provided every time the agent dies while trying to complete a given level of the game.

Once an Ml task is determined, and features of the instances under study are identified, an Ml specialist should select the type of Ml *model* that would approximate the patterns to be found in the data. Examples of such Ml *models* are *decision trees*, or the well-known *neural networks* [67]. Choosing an appropriate model requires expertise and fine-tuning (in particular, tuning of the so-called *hyper-parameters*). As a consequence, failing to obtain meaningful results for a given Ml task may mean that the model is inappropriate, or badly parameterized. We consider that specific considerations on Ml models are beyond the scope of this paper.

3 SEARCH PROTOCOL

For realizing this survey, we used a methodology inspired by Kitchenham [52]. Our protocol relies on two observations. First, the authors involved in this work have different, complementary backgrounds (the two first authors work on Software Engineering and Fv, while the latter is specialized in Ml). Second, no authors had prior knowledge of what could exist in the literature: we were quite certain to retrieve only a few papers, partly due to the opposition mentioned in Section 1. Therefore, we adapted the general methodology of Kitchenham in two ways: we only relied on electronic search queries to collect papers (as we were not sure which academic venues were suitable to such publications, although the study direction we focus on highly suggested to look into Fv venues); and we conducted a pre-study to determine which information is relevant to build our survey. More concretely, we followed three steps:

- First, we queried several search engines to crawl the largest possible set of relevant publications.
- Second, we conducted a pre-study on a set of random papers to determine classification categories for analyzing the literature.

FV	ML
Formal Method ∨ Formal Methods ∨ Formal Analysis ∨ Formal Verification ∨ Model Checking ∨ SAT Solver ∨ SMT Solver ∨ Theorem Proving ∨ Static Analysis ∨ Abstract Interpretation	machine learning ∨ supervised learning ∨ unsupervised learning ∨ semi-supervised learning ∨ clustering ∨ regularization ∨ overfitting ∨ underfitting ∨ feature selection ∨ dimensionality reduction ∨ cross-validation ∨ backpropagation ∨ artificial neural networks ∨ deep learning ∨ support vector machines ∨ kernel methods ∨ decision tree ∨ decision trees ∨ rule learning ∨ fuzzy learning ∨ meta-parameter tuning ∨ hyper-parameter tuning ∨ ensemble learning ∨ ensemble methods ∨ random forest ∨ probabilistic learning ∨ bayesian induction ∨ bayesian probability ∨ reinforcement learning ∨ regression ∨ feature extraction ∨ gradient descent ∨ cost function ∨ data mining ∨ data science ∨ natural language processing ∨ active learning ∨ transfer learning ∨ matrix factorization ∨ manifold learning ∨ multidimensional scaling ∨ preference learning ∨ ranking learning ∨ similarity learning ∨ distance learning ∨ statistical learning ∨ density estimation ∨ text mining ∨ time series analysis ∨ predictive model ∨ learning bias ∨ maximum likelihood ∨ k-nearest neighbors ∨ k-means

Table 1: Query String used for Search Engines. We queried the most popular and well-known academic publishers that offer keyword-based search engines (Elsevier ScienceDirect; Springer Link; Ieee XPlore and Acm Digital Libraries; Semantic Scholar, Scopus, Mendeley and Google Scholar) with the conjunction of strings appearing in Fv and Ml columns.

- Third, all authors reviewed the papers and filled a shared document with the relevant information extracted from the papers.

The rest of this section explains each step in detail, and finishes by formulating our Research Questions in Section 3.4. Table 2 summaries our findings.

3.1 Search Strategy

Having no assumption on how to locate relevant papers, we simply opted for a large list of terms on both sides: we used general-purpose terms for Fv and Ml, strings corresponding to techniques and algorithms, as well as small variations of those terms (e.g., plural and hyphened forms, "-ing" forms of verbs, etc.). The search was conducted between the 10th and the 30th September 2017. We queried the main well-known electronic repositories (Elsevier ScienceDirect, SpringerLink, Ieee XPlore Digital Library, Acm Digital Library, Semantic Scholar, Scopus and Google Scholar), where we manually processed the result pages and selected the relevant publications. We discarded some contributions clearly out of scope based on their abstract and a quick scan of the content. We stopped collecting papers after 10 pages of results for each search engine, because at that point, most results simply correspond to disjunctions of all strings, which becomes highly irrelevant. Our search string is formed as a conjunction of the disjunction of the expressions in each column of Table 1. Finally, at a later stage, we performed a lightweight snowballing from the set of papers we collected, in order to retrieve papers that may have been missed by our search keywords. This resulted in 264 papers collected in a shared online repository.

3.2 Pre-Study & Paper Filtering

The next step aimed at discarding clearly irrelevant papers, and performing a pre-study to extract analysis categories. Each author selected about 20 papers and proposed classification criteria that were collegially discussed. We ultimately retained the elements that constitute a classical Ml pipeline:

(1) *identifying the theme*, i.e. the Fv problem at hand;
(2) *identifying the corresponding Ml task*;
(3) *providing Ml features* to characterize the learning instances;
(4) *figuring out which Ml model (type)* would perform adequately.

Whether extracting Ml model types (Step 4 of the pipeline) from the papers we reviewed has any relevance for readers is debatable: the list we propose is informative, since it only reflects the model types authors have selected, but may in some cases be disputed by Ml experts to be the optimal solution (if such an optimal solution ever exists). Nevertheless, we included this information to reflect the literature, such that readers can grasp what experimentations have been conducted to date for a particular theme/approach.

We then performed a first round of reading in order to roughly classify each paper into Fv approaches, and to discard papers that were clearly out of scope – papers that solely focus on one topic (either Fv or Ml), or papers that leverage dynamic techniques (i.e. that require to actually execute the system). This step resulted in a categorized repository and a shared spreadsheet for cross-checking papers that have unclear contributions. When the Fv contribution was not clear (i.e., whether it does not fit into an Fv approach), we ensured that a cross-check by an author with the appropriate background was made; when the Ml contribution was doubtful (i.e. whether it is really an Ml technique), the author with Ml background checked the paper. This resulted in discarding 96 papers, thus retaining 168 papers for analysis: 53 papers define actual themes whereas the rest define auxiliary resources. In particular, we list in Table 3 papers that provide reference contributions regarding the definition of Ml features.

3.3 Literature Analysis

Once the papers were sorted, the authors with a background in Fv were assigned two Fv approaches to review: they extracted the theme, identified the corresponding Ml task, and retrieved features from each paper. The last author cross-checked the most key papers in each approach to allow a comparison from both perspectives, thus reducing misleading readings about the theme or the task. At later stages when the list of themes became stable, we made vocabulary used throughout the approaches homogenous. When possible we factored out the common features for the approaches (mostly done for Sat and Tp).

3.4 Research Questions

This survey aims at answering the following Research Questions (Rq):

RQ1: *How is Ml used inside Fv tools?* This Rq will be answered in two ways: first, by precisely locating where and how Ml is used inside Fv tools; and second, by providing a higher-level overview that spans over all Fv approaches.

RQ2: *Is using Ml inside Fv tools beneficial?* This Rq is necessary to assess the benefits of Ml in Fv. We answer this Rq qualitatively, based on the assessments made by the authors of the papers we surveyed.

RQ3: *What Ml task(s) is (are) used for which purpose in Fv?* This Rq is intended to associate an Ml task to an Fv theme, as described in Background Section 2.2, and helps bridging both worlds by relating activities in Fv tools to a meaningful category for Ml experts.

RQ4: *Which model types are used to perform the Ml tasks?* This Rq is intended to collect from the reviewed papers the Ml model types the various authors have used to enhance Fv tools. Although indicative and by no means complete or definitive, it provides an interesting panorama of the current practice.

RQ5: *How are Ml features extracted/selected to guide Ml tasks?* This Rq is intended to locate the Ml instances' characteristics as used in the literature, and to eventually list the most common ones.

4 CONTRIBUTIONS

This section catalogs some of the ways Ml complements Fv approaches. We aimed at representativity, i.e. we tried to maximally cover the Ml/Fv complementarities (called *themes* from now on) to propose an overview of the panorama of techniques and current practices in this field. To the best of our knowledge, this is the first study that spans over the main Fv approaches to provide insights on how Ml participates in Fv tools.

We organize this section by Fv approaches in a self-contained way such that each may be read independently. We start with Sat-Smt Solving, then Theorem Proving (Tp), which corresponds to a progression in the expressiveness of the underlying logics (propositional/boolean, then First-Order and Higher-Order). After we introduce and Model Checking (Mc), given the particularity that temporal logic deals with time, and end with Static Analysis (Sa).

Each approach follows the same outline. First, we briefly recall how the Fv approach works. Second, we explain the sources of complexity (typically, NP-completeness) and which countermeasures (e.g. heuristics) have been historically designed to partially overcome them. When necessary, we introduce a brief explanation of the main algorithm supporting the approach in order to fix the terminology and to situate how each theme finds its place in Fv. Third, we introduce for each theme where the Fv/Ml complementarity exists, ground it in terms of Ml task(s), and finally provide examples from the literature. When possible, we indicate the Ml features associated with the theme: when they are common, they are factored out into the section's headers; otherwise, they appear in each particular theme.

In order to guide the reader, Table 2 gathers the highlights of our findings in a comprehensive way: for each theme identified within each approach, we gather all the selected contributions from the literature, the Ml tasks used in that theme and provide hints on the Ml model types that these contributions used.

4.1 SAT / SMT Solving (SAT)

The Sat problem is a decision problem: given a boolean propositional formula, find one (or several) valuation(s) for which the formula evaluates to true. When such a valuation exists, the formula is said to be *satisfiable* (and *unsatisfiable* otherwise). Usually, a formula is processed starting from a canonical representation such as the Conjunctive Normal Form (Cnf), where formulas consist of disjunctions of clauses, which are themselves conjunctions of literals defined as variables or their negation. Theories may enrich propositional boolean formulæ to represent e.g. first-order logic, numbers or richer data structures such as arrays or lists, resulting in the Satisfiability Modulo Theory (Smt) decision problem. Some Fv approaches may be reduced to a Sat problems (e.g., Mc [4] and Tp [14]; for Fv see [77]), making Sat research relevant for Fv.

No algorithm can solve all Sat instances efficiently, which results in a plethora of solving algorithms. Most of the existing algorithms are variations of the Davis–Putnam-Logemann–Loveland (Dpll) algorithm. In practice, tools need to carefully choose the adequate variation for a given (set of) instance(s) to solve them efficiently, generally by reducing the overall runtime. In a simplified way, the Dpll algorithm proceeds as follows: first, chooses a branching literal and assign it a truth value; then, propagates this assignment to other clauses, resulting in unit clauses, i.e. clauses in which only one literal remains unassigned, making its assignment obvious; and finally, propagates those choices appropriately until full assignment, or detection of conflict. When facing a conflict the algorithm backtracks to the previous branching literal to try the opposite assignment. These steps apply recursively until success or unsatisfiability. The Conflict-Driven Clause Learning (Cdcl) improves the general Dpll algorithm by analyzing the cause of the conflicts and backtracking to the appropriate level instead of simply the previous choice, thus improving the overall runtime.

Historically, the Sat community already identified a number of (Sat) instances *features* that characterize the hardness of satisfying an instance. We detail here the most important ones, and mention in the themes the contributions' features that differ from them. SATzilla [100, Fig. 2] integrates a large number of these well-recognized features (around 150): instance size metrics; Variable Incidence / Clause-Variable Incidence Graphs (Vig/Civg) metrics, balance of positive/negative literals in clause and variable occurrences, binary/ternary and Horn clause fractions, number of unit propagation, search space size and other local search probing characteristics. The contributions not explicitly dealing with feature improvements basically reuse different subsets of these features (see e.g., [44, 97] among many others). For 3-Cnf Sat instances, the authors of SATzilla [98] managed to reduce to five the list of the most prominent features, without significant loss of performance. Ansótegui and his co-authors [6] defined new interesting features, targeting industrial Sat instances: the scale-free structure assumes that the ratio of variable occurrences and total number of variables follows a power-law distribution; the community structure measures the modularity of graphs, i.e. how high is a node connected to its direct neighbors; and the self-similar structure measures the fractal effect of CIG and CVIG, i.e. how it changes when a group of nodes is replaced by a single one. They show that relying on those features is computationally affordable, and predicts the instance satisfiability in a way that is comparable to taking all features that SATzilla uses.

4.1.1 Predicting Runtime. Predicting the runtime of an instance (or a subformula) is helpful in many regards: choosing an appropriate solver depending on the runtime; interrupting a computation to switch to another algorithm when the current one takes too long; selecting an appropriate restart strategy when encountering conflicts; selecting a variable to branch on depending on the runtime that will likely result; etc. Runtime is a real, continuous value, making this prediction, strictly speaking, a *regression* ML task. However, the literature often considers runtime *classes* (e.g. long/short runtime of a specific algorithm, i.e. observing whether the runtime crosses a predefined threshold, considering a time budget for solving an instance set). This results in practice in a *classification* ML task.

Horvitz, Ruan *et al.* [44] estimated the runtime of the Quasigroup Completion Problem (QCP, closely related to SAT) by defining a set of features that accurately estimates the resolution progress by reflecting the instance patterns (instance size) and dynamic information about the solver's state (number of backtracks, search tree depth). Samulowitz and Memisevic [85] targeted Quantified Boolean Formulæ and proposed various features: the VSIDS (Variable State Independent Decaying Sum) score, the number of conflicts and the fraction of already solved clauses, the weighted sum between forced literals and VSIDS scores.

4.1.2 Restarting Computations. When encountering a conflict during resolution, the analysis of the literals that led to assignment inconsistencies allows to efficiently backtrack to an appropriate level, avoiding traps that would likely result from a backtrack at another level. A theoretical instance-specific optimal restart strategy exists, but requires the knowledge of the instance's runtime distribution (rarely known and difficult to compute) [66]. Those strategies are split into two categories: *universal* strategies are defined independently of the instance; and *dynamic* strategies adapt to the search length, i.e. the number of already assigned literals. Since no strategy outperforms all others on all datasets [45], adapting the strategy to a dataset is often better, and constantly reevaluating the portfolio of restart strategies is desirable to keep improving SAT solvers performances [13].

For solving as many instances as possible within a given time budget, selecting the best restart strategy on an instance basis (according to its features) represents a *classification* ML task: from a set of predefined strategies, determine which one would be the best to optimally (i.e. by minimizing the expected runtime) solve the instance at hand. For very hard instances (typically, industrial/crafted instances representing cryptographic or planning problems [6]), or for solving several instance sets while minimizing the overall runtime, switching between strategies during the solving is often more efficient, when some strategies are known in advance to be the most powerful for the given instances (sets). This can be seen as a *reinforcement learning* problem, in which restart strategy choices are seen as the possible *actions* to reinforce. Rewards are defined differently depending on the particular contributions.

Haim & Walsh [39] proposed to select a strategy from a portfolio of 9 that were proven to perform well on at least one dataset. The training is based on a subset of features taken from [100]; the prediction is realized dynamically, while solving the instance. Horvitz, Kautz, Ruan and their colleagues adopted a more contextualized approach, in the context of dependent runs [50, 81]: they used predictors (called observators) on some instances of a set to help determine how to perform restarts for the other instances of the set. Observators are generally classifiers trained on a few instances that predict whether (future) instances would be satisfiable or not, or whether their runtime takes a short/long time.

Nejati *et al.* [74] targeted cryptographic instances, whose runtime is usually much longer than other instance types, making restart strategies a core component. Solving an instance requires, during the solving itself, to change / switch strategies, among the ones that are known to be the most effective in the literature: uniform, linear, luby and geometric [13]). Reinforcement Learning is performed through the following steps. First, a strategy is chosen, and the general algorithm proceeds with the solving until the strategy imposes a restart. At this point, the strategy is rewarded based on the average Literals Block Distance (LBD) of the learned clauses generated since the strategy was selected. Finally, this results in choosing/favoring strategies that produce small LBDs for the future solving steps. Gaglio and Schmidhuber [35] considered the problem of using the best restart strategies for a set of instances to minimize the global runtime. They choose between two strategies (Luby's universal and uniform). After one step of solving an instance, they reward the strategy that results in a runtime that stays close to the runtime of the previous instances. This results in favoring the strategy that provide a global runtime for the set that is the closest to the runtime of most of the instances in the set.

4.1.3 Selecting the Branching Variable. Choosing the most appropriate branching variable is crucial for improving solvers' runtimes, because it ultimately minimizes backtracking (which can be seen as a step back towards a solution, since it implies unassigning some of the already selected variables). This can be seen as a *reinforcement learning* problem: along a SAT instance solving, choose the next variable to branch on, such that the reward attached to the variable choice, called *score*, maximizes the progress for solving the instance. Note that this task is *non-stationary* from the SAT solving viewpoint: after each choice, a variable cannot be selected anymore unless a conflict occurs.

Liang *et al* [60, 61] explored two different reward computations based on different branching heuristics: in [60], they used a conflict-history-based heuristic for variable selection; while in [61], they used another heuristic called learning rate branching. Both rewarded the generation of learned clauses locally, at each step of the solving. Later on, Liang *et al.* [62] rewarded selections that maximize *global* branching learning rates, i.e. rates for the whole solving. Fröhlich *et al.* [33] penalize the candidate variables choices that minimizes the number of unstatisfied clauses. Lagoudakis and Littman [58] penalize branching rules (chosen among seven known as the best working) whose solving time is too long.

4.1.4 Determining Best Solving Algorithm. The SAT community identified families of instances that may be better solved with specific algorithms, enhancing specific criteria (mostly, solving runtime). In a pure form, this is a *classification* ML problem: from a set of instances, determine which solving algorithm(s) would be the most efficient according to a given criterion. However, this could be seen as a *regression* ML problem, when the goal is to predict an algorithm continuous probability of success. Both ML tasks are achieved offline, i.e. before running the solving algorithms. Note that *pre-solving* is mainstream, i.e. trying some predetermined, quicker algorithms that may solve some of the instances, leaving the portfolio selection to focus on difficult instances.

SATzilla [100] is one of the best portfolio solvers [59]: it relies on a large variety of specialized SAT algorithms that are chosen according to the specificities of the instance at hand, based on 115 features (cf. feature discussion in Section's header). It allows to switch to another algorithm ("next best match") when the one attributed initially takes too long. AutoFolio [63] selects algorithms based on features similar to the ones in SATzilla.

4.1.5 Configuring SAT Solvers' Parameters. Instead of setting default values for the multiple parameters of the various SAT algorithms constituting a portfolio, many SAT solving tools choose to expose those parameters to the end-users, passing them the burden of configuration. The end-user faces a highly difficult task: which parameter settings of the algorithm(s) perform best on a set of instances, minimizing a cost function (typically in SAT, runtime). Several approaches already exist based on heuristics, but the domain recently gained attention with ML. This particular domain has its own competitive event: the Configurable SAT Solver Challenge. Finding the optimal values of a SAT solver's parameters is a *r*egression or a *c*lassification task whether if a continuous or a categorical value is predicted.

Hutter, Hoos and Leyton-Brown [46] defined SMAC (Sequential Model-based Algorithm Configuration), a technique and tool that generalizes the classical optimization algorithm by using training, based mostly on SATzilla's features. AutoFolio [63] parametrizes ClasspFolio 2 (the default version of SATzilla'11), resulting in a highly parametrized algorithm framework.

4.1.6 Learning Satisfiability. Tackling the whole SAT problem through ML seems difficult, but has been partially attempted. Overall this corresponds to a *classification* problem (although many other ML tasks are performed in between to serve the main goal): determine whether a SAT instance is satisfiable or not.

A partial prediction on a subformula may guide a solver for other tasks (e.g., restarting efficiently, or evaluating the potential of a variable selection, among others). For example, Wu [97] predicted 3-CNF instance satisfiability with seven features from SATzilla and other classical ones. They reuse the partial prediction to determine which value is preferable for a branching literal. Although the technique is applied to subformulæ, nothing prevents the technique from being used on a larger scale (although optimizations are naturally expected). One would expect that a SAT problem becomes harder when the number of clauses increases. In fact, the most difficult instances are those whose ratio of clauses over variables is near the so-called *phase transition* (particularly for 3-CNF instances): the number of clauses and variables is at equilibrium, making instances neither underconstrained (i.e. exhibiting many possible solutions), nor overconstrained (i.e. exhibiting many contradictions). Devlin and O'Sullivan [29], and later on Xu, Hoos and Leyton-Brown [99] studied several classifiers for predicting the (non-)satisfiability of general SAT as well as 3-CNF instances, based on the classical features used in SATzilla. Xu, Hoos and Leyton-Brown tried to minimize the number of necessary features to build good classifiers, and managed to reduce to two features while staying robust comparing to classifiers with more features.

4.2 Theorem Proving (TP)

When the semantics of a software or of a system is expressed as mathematical theories, verification conditions for those systems can be formulated as mathematical properties of those theories. In FV, Theorem Provers (TP) are then employed, in a more or less automated fashion, to prove or disprove such properties.

Mathematical theories are composed of mathematical facts, which are assumed to be true. TP is used to infer new facts about the theory, using the inference rules associated to the logic of choice. In this sense, a mathematical statement (known as conjecture) becomes a theorem if it logically follows from the theory. More precisely, TP operates as follows: 1) it receives as input a set of facts from a mathematical theory which are assumed to be true and a *conjecture*; 2) it performs a number of inferences on those facts using the set of rules that describe the semantics of the logic being used; and 3) outputs a proof for the conjecture or a trace thereof, if one exists, in which case the conjecture becomes a *theorem* and can be added as a new fact to the theory.

First-Order Logic (FOL), one of the most popular logics in TP, is semi-decidable. The bulk of the work of applying ML techniques to TP has thus targeted FOL. ML techniques assist or replace existing expert knowledge-based heuristics in order to better navigate the border between decidability and non-decidability and more efficiently lead theorem proofs to completion. Because they present a high level of automation, provers for FOL are called *automated theorem provers* (ATPS).

Higher-order logic (HOL) adds to FOL the quantification of predicate and function symbols. Such expressiveness is convenient to express verification problems that would otherwise be too difficult or impossible to express in lower-order logics. However, HOL is undecidable and presents fewer opportunities for automation than FOL, which means parts of the proofs need to be guided by humans. For this reason, in the context of HOL, provers are called *interactive theorem provers* (ITPS).

Decidability and efficiency issues in ATPS/ITPS mean that decisions are delegated onto humans at many points of the proof. Such decisions involve for instance: choosing facts (also known as premises [57]) from the theory relevant to the proof at hand; picking sets of proof engine parameters (also known as *heuristics* [19]) such as for instance sets of inference rules used [56]. It is in supporting or replacing the human in these decisions that ML comes to the aid of theorem proving.

Features used to characterize facts or conjectures about theories are majoritarily the symbols found in those mathematical statements [69], for example literals or predicate names. Metrics such as the number of clauses, literals or subterms, or yet specific metrics about the translations of logical formulas into normal forms have also been used [56]. Other authors have attempted to use types [47], or meta-information about the theory name and its presence in various mathematical databases [55]. Recently, Kaliszyk and his co-authors have proposed features that capture semantic relationships between mathematical statements [49].

4.2.1 Selecting Facts. Which subset of facts to take from a large theory in order to complete a given proof as efficiently as possible (or at all) is one of the most prominent applications of ML to TP [57]. Fact selection is a *classification* task: either a fact is relevant for the current proof or not, potentially with a probability reflecting a level of certainty.

Fuchs [34] uses data from previous proofs to train a model for computing the usefulness of the available facts for the next proof step. Alama [1] preanalyzes a large mathematical repository of formalized theories in order to calculate dependencies between parts of those theories that can then be used at proof time. Kaliszyk *et al.* [16, 47] use classification models to rank facts in HOL theories according to their assumed relevance for the proof of the conjecture. They then reduce the best ranked of those facts to simpler problems that can be handled by fast FOL ATPS to help in parts of the proof of the original conjecture. Again Kaliszyk, together with his co-authors, provide in [48] a compelling account of how MALAREA performs ML-based fact selection for the equational reasoning ATP E [86] beating the competition in large-theory contests. Alemi *et al.* report in [2] the first application of deep learning to fact selection for large theories, concluding that neural networks do help in large-scale automated reasoning without requiring hand-crafted features. They mention nonetheless that hybrid premise-selection solutions where hand-crafted solutions are used together with their solution may yield even superior results. Loos and her colleagues confirm this thesis in [64] and conclude that fact selection mixing neural networks and other methods is particularly useful for hard theorems that require complex reasoning.

4.2.2 Configuring Proof Engine Parameters. Proving a particular conjecture is typically achieved more or less efficiently (or at all) by providing the prover with a set of parameters. It is well established in the TP community that certain parameters configurations are better suited for the proof of certain classes of conjectures [19]. Constructing such parameters automatically for specific conjectures is assisted by an ML *regression* task: ML helps in predicting proof runtime when heuristics are evaluated on specific conjectures.

Kühlwein and Urban investigate in [56] a method to automatically tune parameters of ATPS in order to optimize proof times . Their method starts from a set of random or predefined heuristics and the ML algorithm learns to predict how fast these heuristics perform on classes of existing problems. The technique then iteratively improves the parameters of successful heuristics by using the prediction learner.

4.2.3 Selecting Pre-Defined Proof Engine Parameters. Some theorem provers select proof engine parameters that were manually or automatically generated (cf. *Configuring Proof Engine Parameters* theme). This is a *classification* ML task: given a conjecture, provide the heuristic that will most likely produce a proof for it in an efficient manner.

Bridge [19], a reference for this theme, evaluates more than fifty features and concludes that only combinations of very few features (up to two) are required to build classifiers that vastly outperform random proof engine parameter selection. With his colleagues [20] Bridge later confirms that their system yields the same performance as E's internal proof engine parameter selection mechanism, without requiring the introduction of any human-expert knowledge. Additionally, the system is also able to decline some proofs in case no proof engine parameters can lead to the completion of the conjecture's proof in an acceptable amount of time (or at all). The authors report that declining proofs greatly improves performance, while only moderately reducing the amount of provable theorems.

4.2.4 Guiding Interactive Proofs. Due to the undecidability of HOL, there is no systematic way of finding proofs for conjectures in such logics. ITPS such as COQ [68], ISABELLE [15] or MIZAR [38] are used to assist the mathematician in proof finding, while PROOF GENERAL [8] provides a high-level user-friendly interface to those ITPS. ITP environments can act as recommender systems to suggest for example promising facts to be used in the proof. Because this theme touches many parts of proofs, both *classification* and *clustering* ML techniques can be used. *Clustering* becomes particularly useful here as it can inform the user of potential next steps through statistical analysis on similar proofs – it however cannot lead to automatic decision making such as when supervised approaches are used.

Urban [93] describes a set of proof aids in EMACS for MIZAR, explaining how ML classification algorithms are used to suggest facts to a mathematician for the continuation of the proofs. Mercer *et al.* [31] propose a system and a user interface for recommending the next proof step, based on Duncan's work [30] on modeling proofs with Variable Length Markov models. Komendantskaya and Heras interface PROOF GENERAL with back-ends running clustering algorithms [53] to gather statistics on data from previous proofs.

4.2.5 Learning Theorem Proving. Rocktäschel and Riedel attempted to learn the backward chaining algorithm for FOL [80]. Starting from a set of neural networks that modularly perform generic TP-related tasks (such as for example *unification*), the authors propose an algorithm that is able to assemble those modules in order to deduce new theorems from a given knowledge base. This contribution has the particularity that, due to the fact that modules are used, the proof is in itself interpretable – more specifically how those modules are used during the proof provides a proof trace. As the output of the neural network is a proof score that describes the confidence in the derived facts, we technically classify it as a *regression* task.

4.3 Model-Checking (MC)

Model-Checking (MC) [24, 78] consists of abstracting the concrete system's execution into a finite-state automaton (that can be extracted automatically from the program, and whose execution may be infinite); and the properties of interest are expressed in temporal logics. The model-checking procedure explores exhaustively the (abstract) state space, and either validates the properties, or returns a counterexample (that may be a false alarm). In practice, exhaustive exploration is difficult: many techniques were crafted to reduce the state space (symmetry, slicing, partial evaluation, to cite only a few), or enhance its exploration (through path exploration heuristics). They nowadays equip most tools. We did not find Ml contributions that work at the Mc algorithm level like for others Fv approaches; rather, we found contributions that help detecting counterexamples faster, or reducing false positives (using the well-known Cegar approach). Note that we did not include contributions for Assume-Guarantee Reasoning [3, 10, 21, 27, 36, 72, 75, 91] based on the L^* algorithm [5, 79]: it is difficult to conclude without deepening the subject whether L^* is an Ml algorithm, and which Ml task it corresponds to.

4.3.1 Finding a Counterexample. Model-Checkers should be oriented towards *error detection* [26]: they should favor the discovery of errors rather than focusing on guaranteeing correctness. As a consequence, optimizing counterexample finding is crucial. A possible approach is to explicitly *guide* the state space exploration towards paths that may favor such counterexamples, based on the property of interest at hand. This may be achieved through *reinforcement learning*: a reward favors positive paths for invalidating the property; while a punishment discourages negative paths validating it. Note that the qualificatives *positive*/*negative* correspond to the *error detection goal* instead of the traditional Mc goal.

Araragi and Mo Cho [7] targeted the production of counterexamples for *liveness* properties that represent responses, i.e. a (premise) event is expected before a (response) event should eventually occur. The authors kept track of the premise occurrence at the state space level, and rewarded explorations that stayed on paths between premise and response as long as possible. This would lead to cyclic, or very long (or infinite) paths that would invalidate the property. Behjati, Sirjani and Ahmadabadi [11] studied Ltl properties on Büchi automata with on-the-fly Mc: the reinforcement learning agent is punished when following non-accepting cycles, and rewarded when finding unfair accepting cycles, until a fair one is found, leading to the property's invalidation.

4.3.2 Refining Abstraction based on Counterexamples (CEGAR). A spurious counterexample (false alarm) happens when the last state of a path in the concrete system mixes both *deadend* states, i.e. states with no concrete transition to the failure state; with *bad* states, i.e. states using (system) variables useful to prove the property, that are not taken into account, and abstracted together with deadends. To eliminate such a counterexample, some variables need to be identified and become visible, i.e. separated properly within the abstraction. This is known as the *separation problem* in Cegar Mc, which is a *classification* Ml task: given a (sub)set of system variables from the failure state, determine whether they should be classified as *deadend* or *bad*. This information then allows for an abstraction refinement that, even if not optimal, makes it possible to discharge the counterexample. Note that for realistic systems, enumerating all the variables is impossible: a preselection is generally operated beforehand.

Clarke, Gupta and their colleagues [23, 25] implemented this technique for model-checking hardware circuits, training a learner on samples automatically extracted from the concrete system.

4.3.3 Extracting Most Common Error Patterns. Concurrency errors often result from the same error types [65, 82]. Finding the recurring paths or rules leading to such errors is related to the Ml task of *frequent item set finding*.

Pira, Rafe and Nikanjam [76] characterized frequent patterns as sequence of rewriting rules in Groove, a graph-based model transformation tool, using a variation of the APriori algorithm. Those patterns are discovered on smaller systems with similar architectural design, then used to guide Mc on larger systems.

4.4 Static Analysis (SA)

Static Analysis (Sa) designates a large panel of techniques aimed at computing any information about a program, generally directly on its Abstract Syntax Tree Ast. The underlying abstractions rely on predefined approximations (possibly parameterized by users' inputs): this results in a fixed set of properties of interest, e.g. extracting Android apps' permissions from `.apk` files, which may be parameterized to find only device-specific ones. In most cases, the analysis does not carry in itself its own final usage: for instance, permissions, in themselves, do not give any information about an app being a malware. Most of the contributions in Sa leverage Ml to bridge this gap, by trying to find links, or reccurent patterns, in the retrieved information, e.g. malwares are apps that present significant discrepancies between exhibited permissions and actual executed code.

4.4.1 Identifying Actionable Alerts. Sa tools often issue large amounts of alarms that warn about code style violations, trivial defects with no impact on functionalities, false positives and, of course, real bugs. Too many warnings hamper developers' productivity by forcing them to review alarms, diverting their attention from issues that matter (cf. surveys on alarms handling [43, 71]). Reducing and classifying those alarms based on previous iterations/similarity significantly enhances the experience of using Sa tools. This is a *classification* Ml task: from a set of flagged alarms, which ones are *actionable*, i.e. require a specific bugfix from a developer.

Heckman & Williams [42] postulated that characterizing whether an alarm is actionable highly depends on each project and developers involved. They reduced the unactionable alarm number by first gathering alarms and their features, then by selecting the most relevant ones for training a classifier for future projects. They used as features the usual metrics (LoC, etc.) with code change history, and alarm types (null pointers, etc.) with alarm severity delivered by Sa tools. Hanam *et al.* [40] proposed to classify alarms as actionable or not by identifying recurrent patterns based on characteristics related to code statements: invocation and creation sites, field access, binary operations, catch statements as well as other various structural features like method signatures and class names. They showed that patterns effectively exist and help discover more errors than classical Sa tools reports. Kremeneck *et al.* [54] correlated alarm reports with their code localization to classify alarms raised at later stages of code integration. Ruthruff *et al.* [83] identified the legitimate alarms that are more likely to be acted on by developers, based on Ml features similar to [40, 42]. They also managed to reduce the number of metrics necessary for performing the classification, while preserving a correct classification ratio.

4.4.2 Predicting Bugs from Previous Code Versions. Instead of running Sa tools during the development phase, an interesting line of research consists of predicting, at code submission into a repository, whether a code change likely contains bugs, based on the analysis of previously submitted changes in a project. This presents several benefits: the change is still fresh in mind, and several actions may be taken (from code review, testing, to Fv techniques), targeting the recent change. This is a *classification* Ml task: predict whether a code change is likely to contain bugs, based on the analysis of previous code version(s).

Kim and his colleagues [51, 88] as well as Hata *et al.* [41] introduced change classification by analyzing the change history, based on log messages keywords and correlations to bug fix requests. Their predictors rely on various features: change metadata on the versioning system, complexity metrics and various Sa information to locate code change and analyze their impact. Kim and his colleagues also investigated in [51] the possibility of reducing the large amount of features extracted from change history.

4.4.3 Classifying Android Apps as Malware. With hundreds of new apps and countless updates, detecting malware in Android apps has become crucial to ensure end-users' security. Most contributions mixing Ml with Sa rely on *misuse detection* that flags an app when permissions mismatch the actual app functionalities (cf. [73] for an overview of static and dynamic malware detection). This qualifies as a *classification* Ml task: given an Android app, together with a set of characteristic features that are extracted statically, determine whether the app contains a malware. The listed contributions differ in features and training data sizes for training: we comment on features and refer to each paper for other details.

Aung & Zaw [9] used five characteristic permissions (internet access, configuration files change, send/write Sms and phone calls) extracted directly from the distribution files (.apk) of known malware and goodware from classical Android Markets. Sahs & Khan [84] used a combination of permissions, categorized as built-in (like accessing Internet) and non-standard (like accessing the camera or the localization), paired with Control Flow Graphs to analyze the app's code. Yerima and his colleagues [102, 104] (cf. [103] for details) extracted a set of complementary features, characterized by specific keywords: a total of 125 permissions extracted from the manifest; features related to Linux commands hidden in compiled or library code, used for escalating privileges or launching scripts and malicious binaries; and standard Android Api calls extracted from the app's Dalvik code, to detect required interactions with the various devices (e.g., Sim, location or network accesses, device or user ids, or method invocation and class loading, boot process, etc.) or to enrich apps with various functionalities (e.g. contact, Sms or Url lists already accessed).

4.4.4 Learning SA. Tackling the Sa problem itself, directly from the source program is a *classification* Ml task: providing the Ast, does the (fixed, predefined) property hold or not.

Several authors attempted this [22, 70, 101] for various analyses, but all noticed that, for the approach to scale, sufficient training data for each property needs to be available (positive as well as negative training, i.e. verifying *and* falsifying the property at hand).

5 DISCUSSION

Is there a love story between Fv and Ml? In this paper, we surveyed one direction of this relationship: how Ml contributes to enhance Fv activities. Without being exhaustive, we have shown throughout Section 4 that Ml enhances all the spectrum of Fv approaches (Static Analysis, Model-Checking, Theorem-Proving, but also Sat/Smt-Solving) at different levels, using different techniques and for different purposes.

How can Fv and Ml experts collaborate to leverage Ml's current practices in order to enhance and improve current Fv tools? We follow the classical Ml pipeline: (i) *identifying the Fv problem* at hand; (ii) *identifying the corresponding Ml task*; (iii) *providing Ml features* to characterize the learning instances; (iv) *figuring out which Ml model (type)* would perform adequately.

The remainder of this section discusses each of these points, gives general perspectives on what seems promising for the future, and provide answers to the research questions formulated in Section 3.4.

*FV Problems (**RQ1:** How is Ml used inside Fv tools?)* Fv experts first identify what they expect to improve, compute, or which kind of pattern they seek in their data. This is one of the topics covered by this paper: each of the themes indicates precisely to which extent Ml is used in the overall Fv approach: some contributions/tools invoke Ml at various steps, or even handle the approach altogether. We noticed several patterns according to Ml categories. *Supervised techniques* are often associated with two kinds of usage observed in Sat and Tp: *external* guidance, which stands for Ml models that choose an appropriate heuristic, strategy or algorithm (e.g., portfolio solving in Sat and proof engine parameter selection in Tp); and *internal* guidance, which represents situations where Ml models play the role of heuristics/strategies, by selecting the next following step in a more global algorithm (e.g., fact selection in Tp or restart selection in Sat). Other uses do not fit these categories. For instance, the *interpretative gap filling* in Sa is performed through Ml models by finding links or recurrent patterns in the collected information (e.g. the link between permissions and the malware/goodware classes).

Unsupervised techniques do not prescribe an "ideal" answer, but rather try to identify general patterns. The resulting tasks (mostly clustering) seem more adequate for Tp, the only Fv approach favoring interactivity. However, it is not excluded that unsupervised techniques may bring new insights into other Fv approaches, even those that already are fully automated.

*ML Tasks (**RQ3:** What Ml task(s) is (are) used for which purpose in Fv?)* Once the Fv problem is identified, Fv specialists associate an Ml task to guide Ml experts towards the right set of techniques and models. When Ml contributes to a small portion of the Fv process we have often observed *classification* tasks, i.e. selecting a candidate artifact among several available ones. In Tp and Sat, selecting heuristics according to some criteria (potential for proof completion; solving runtime) among those already programmed by experts, relieves Fv users from the burden of having to maintain explicit knowledge of those heuristics. Such approaches have improved tools significantly, as witnessed by Tool Contests in Sat and Tp. In Mc and Sat, *reinforcement learning* is used as a way to "guide" the tool towards a counterexample and the most promising branching variable. The final Ml task may differ according to the experts' viewpoint, but is ultimately guided by Fv experts' needs: for example, a *regression* task such as predicting Sat solvers' parameters values, or runtimes in Tp, may very well be turned into a *classification* task by imposing runtime thresholds.

	Themes	Contributions	ML Task	ML Model Types gathered from Contributions
SAT-SMT	Predicting Runtime	[85](+) [44](-)	Classification (or Regression)	Logistic Regression ; Decision Trees
	Restarting Computations	[39](o) [50, 81](+) [74](+) [35](o)	Classification Reinforcement	Logistic Regression ; Decision Trees Multi-Armed Bandit
	Selecting Branching Variable	[60, 61](+) [62](+) [33] [58](+)	Reinforcement	Multi-Armed Bandit ; Temporal Difference
	Determining Best-Solving Algorithm	[100](+) [63](+)	Classification	Logistic Regression
	Configuring Solvers' Parameters	[46](+) [63](+)	Regression	Support Vector Machines ; Random Forest Regression
	Learning Satisfiability	[97](+) [99](+) [29](+) [87]	Classification	Logistic Regression ; Decision Trees ; Random Forests k-Nearest Neighbors ; Naïve Bayes ; Neural Network
TP	Selecting Premises	[2](+) [64](+) [34](+) [1](+) [47](+) [16](+) [48](+)	Classification	Kernel-based models ; Naïve Bayes ; k-Nearest Neighbors
	Configuring Proof Engine's Parameters	[56](+)	Regression	Kernel-based models
	Selecting Predefined Proof Engine Parameters	[19](+) [20](+)	Classification	Gaussian Process Classifiers; Kernel-based models
	Guiding Interactive Proofs	[93](○) [31](○) [53](○)	Classification / Clustering	Naïve Bayes ; Variable Length Markov Models / k-Means
	Learning Theorem-Proving	[80](-)	Regression	Neural Networks
MC	Finding Counterexamples	[7](+) [11](+)	Reinforcement	Q-Learning
	Refining Abstractions	[23, 25](+)	Classification	Decision Trees
	Extracting Most Common Error Patterns	[76](o)	Frequent Item Set	A-Priori
SA	Identifying Actionable Alerts	[42](○) [40](+) [54](+) [83](+)	Classification	Decision Trees ; Bayesian models ; Logistic Regression ; Rule-Based
	Predicting Bugs from Previous Code Versions	[51](○) [88](+) [41](+)	Classification	Support Vector Machines ; Bayesian models
	Classifying Android Apps as Malware	[9](○) [84](○) [102](+) [104](-)	Classification/Clustering	Decision Trees ; Random Forests ; Support Vector Machines ; Bayesian models / k-Means
	Learning SA	[22](-) [70](+) [101](○)	Classification	Neural Networks

Legend: (+) improves the state of the art; (-) comparable to or worse than state of the art; (○) no information on how the approach relates to the state of the art

Table 2: Contributions and ML tasks related to each theme within each FV approach.

	Reference Papers
SAT-SMT	[98, 100] [6]
TP	[69] [47, 49] [55] [56]
MC	[23]
SA	[40, 42] [88] [41] [9] [84] [104]

Table 3: Main papers defining ML features.

*ML Features (**RQ5:** How are ML features extracted/selected to guide ML tasks?)* The choice of features is critical for the learning process. We noticed two main categories of such choices in FV: features are either based on the *raw input* used for the FV approach (e.g., AST and manifests in SA; CNFS in SAT and TP); or based on *measures* computed on those raw inputs (e.g. Call Graphs for SA, ratios and clause numbers for SAT). Those features were identified experimentally while tuning heuristics and/or trying to improve existing algorithms, and often predate the introduction of ML. Identifying the appropriate features is the task of FV experts, but using ML may help enhancing them or filtering out superfluous ones. Table 3 lists the main contributions in each FV approaches.

*ML Models (**RQ4:** Which ML model type are used to perform the ML tasks?)* The selection of an ML model to perform a specific ML task is the final step. This is extremely delicate, and essentially a problem for ML experts. We noticed in the surveyed contributions that the use of specific models by FV experts is not always clearly motivated. In fact, the literature suggests that the ML model is often selected among the set of those available in the ML tool(s) the authors are familiar with (e.g. the Weka Workbench [32]). This is not incompatible with common practice: ML experts often use a trial-and-error approach to determine which model performs best for a given task. Having bad performance with a specific ML model does not always imply that the model is not suited for the ML task, but rather it is not optimally parameterized. It is sometimes impossible to figure out in advance which model will work best (as a consequence of Wolpert and McReady's No Free Lunch Theorem [96]). However, we believe that identifying precisely the answers to the previous steps should provide ML practitioners with sufficient information such that they can exercise their expertise.

__RQ2:__ Is using ML inside FV tools beneficial? Table 2 presents a summary of our findings: for each theme inside each FV approach, we list the contributions we reviewed and indicate whether the results of each contribution has improved the existing state-of-the-art, and points to the models commonly used by all contributions in a theme. From a statistical viewpoint, contributions that claim to have brought improvements largely outnumber the ones with similar or lower quality than state-of-the art.

Towards end-to-end FV. All in all, we observed a pyramidal use of ML models. On one end of the spectrum, ML models are used in a very narrow fashion inside FV tools for solving very specific problems inside tools. For instance, in SAT or TP, the structure of the current resolution algorithms can be preserved while delegating onto ML models the optimization of specific choices that are traditionally handled by heuristics (like restarts or fact selections). On the other end of the spectrum, we noticed several attempts to handle an FV approach globally: for instance, predicting satisfiability of a formula [97, 99], building a proof [80] or learning static analysis directly [22, 70, 101]. Between these two ends, a range exists determined by how much FV expert knowledge is considered while solving the FV problem. ML typically aims at finding objective generalizations; however, if injecting domain knowledge (e.g. on how current FV algorithms are designed) significantly improves performances, it becomes meaningful to integrate it to relieve the ML algorithms from struggling to learn specific aspects while allowing them to focus on more global aspects of the problem. Our survey points to that fact that the use of ML for specific FV tasks is over-represented, at the expense of more recent holistic strategies.

Such holistic strategies have already radically changed fields such as image and natural language processing, especially after the introduction of Deep Learning [37]. It became clear from our literature study that FV tools that introduced ML in their workings started to deliver impressive performances in international contests (e.g. [20, 64] in Tp; [63, 100] in Sat). However, using more powerful Ml models directly hampers the interpretability of their results [12]. This is problematic for Fv, since its techniques are often used to ensure the correctness of safety-critical software for which human-understandable justifications need to be provided. We strongly call for a more systematic review of the domain in order to precisely identify future directions in this research domain. We believe *machine learned* Fv is potentially achievable when sufficient amounts of data will be collected, just like image analysis for medical diagnostic [94], board game playing [89, 90] or even self-driving cars [17, 18]. While a decade ago progress in such domains seemed extremely difficult, it has now become reachable for (state-of-the-art) Ml. Rather than following designs and abstractions created by humans, Ml may indeed find fresh new ways of handling Fv problems, opening the potential to entirely reshaping the Fv domain.

6 THREATS TO VALIDITY

Although we designed our search protocol in a very inclusive way, as witnessed by the number of papers finally discarded manually, we may have missed some relevant contributions. First, we relied exclusively on repository search, whereas crossing searches with top-venues both in Fv and Ml may have brought new interesting contributions. Second, we stopped searching online repository after 10 result pages Third, we performed the search in September 2017, before many important Fv, as well as Ml, venues take place. To mitigate these points, we have conducted a backward snowballing on the main papers in each approach (typically, the most cited ones) in January 2018, and looked at the program of some of the relevant top venues. In future revisions of our survey, we will integrate forward snowballing, as recommended by Wohlin [95], by looking *a posteriori* at top-venues in recent months.

Furthermore, our survey is likely to have missed gaps in the literature, meaning that additional themes and/or better relations between themes, tasks and Ml models may surface in future work. In fact, as research progresses in this area and Ml becomes more widely adopted, we expect to find new themes that are for the moment not explored by the community: this survey may well be seen as a current snapshot of the available contributions in the domain, rather than a definitive survey that closes the matter.

In Table 2, the information regarding state-of-the-art improvements (noted as +/o/-) has been collected from each contribution relying solely on the article's text. We have taken into account the comparisons with other tools operated by the authors, or analyzed explicit statements from them on how their method/tool compare to others. We have been particularly attentive to available data on relevance (precision and recall) and performance (speed). These comparisons found in the literature form a heterogeneous set: some authors compare their work with solutions where no Ml is used, whereas others provide comparisons with Ml-based tools; datasets which are used as the basis for learning are often small and have not been made available online, meaning reproducibility of the presented results is, in general, not possible. In some articles, no comparison with the state of the art is provided by the authors: in some instances, this means that the Ml technique addresses a problem that was previously manually handled, or not handled at all; in others, this simply means no comparison is provided by the authors.

Finally, the Ml model types presented in Table 2 have been gathered for informational purposes and not in an exhaustive way. In this sense, the association between themes and Ml models does not imply an exclusive relation of appropriateness between them.

7 CONCLUSION

In this paper, we have explored how Ml contributes to enhance Fv tools efficiency. We covered four classical approaches, namely Sat, Theorem-Proving (Tp), Model-Checking (Mc), and Static Analysis (Sa), for which we catalog a list of themes, i.e. precise points where an Fv problem is translated into an Ml task to be handled by an Ml model type. Although preliminary, our survey shows not only that Ml may keep on contributing to the Fv field in both short and medium terms. It also shows that integrating Ml methods inside Fv tools is largely beneficial, as demonstrated in Sat and Tp tools that regularly achieve new scales of efficiency. However it is still essential to tackle challenges that were until recently thought as unreachable (e.g. attacking realistic cryptography protocols like Rsa).

By studying the intricate relation between Fv and Ml over a large spectrum of approaches, we were able to frame the way Fv and Ml experts collaborate in a classical Ml pipeline: identifying the Fv problem (corresponding to our *themes*); determining the corresponding Ml task; providing Ml features; and figuring out which Ml model type is the most adequate for the task. We also captured general trends, the most challenging being *learning* Fv approaches on their own, as witnessed by many attempts in e.g. Sat, Tp and Sa.

The reverse direction of the love story has been left untouched, despite a recent growing interest: how can Fv may help Ml. Verifying Ml tasks results has nowadays become a stepping stone in the adoption of thrilling new technologies: for example, correctly identifying road signs directly influences the behavior of self-driven cars, which in turns guarantees the safety of passengers. One of the main reasons that make such verification hard is that the implicit models (e.g. neural networks, one of the currently most promising learning technologies) are difficult to grasp and understand for humans. Properly stating what kind of properties one expects from such implicit models is even more difficult. Building appropriate abstractions of such models, that often integrate probabilistic and/or continuous computations is a key challenge. Therefore, specifying what models to accept appears to be difficult.

REFERENCES

[1] J. Alama, D. Kühlwein, E. Tsivtsivadze, J. Urban, and T. Heskes. Premise selection for mathematics by corpus analysis and kernel methods. *CoRR*, 2011.
[2] A. A. Alemi, F. Chollet, G. Irving, C. Szegedy, and J. Urban. DeepMath — Deep Sequence Models for Premise Selection. *CoRR*, 2016.
[3] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Computer Aided Verification*, pages 548–562, 2005.
[4] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. McMillan. An Analysis of SAT-Based Model Checking Techniques in an Industrial Environment. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 2005.
[5] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[6] C. Ansótegui, M. L. Bonet, J. Giráldez-Cru, and J. Levy. Structure features for sat instances classification. *Journal of Applied Logic*, 2016.
[7] T. Araragi and S. Mo Cho. Checking liveness properties of concurrent systems by reinforcement learning. In *Model-Checking and Artificial Intelligent*, 2006.
[8] D. Aspinall. Proof General: A Generic Tool for Proof Development. In *TACAS*, pages 38–43, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.
[9] Z. Aung and W. Zaw. Permission-based android malware detection. *International Journal of Scientific & Technology Research*, 2(3):228–234, 2013.
[10] H. Barringer, D. Giannakoupolou, and C. Pasareanu. Proof rules for automated compositional verification through learning. In *Workshop on Specification And Verification of Component-Bases Systems*, pages 14–21, 2003.
[11] R. Behjati, M. Sirjani, and M. N. Ahmadabadi. Bounded rational search for on-the-fly model-checking of LTL properties. In *FSE*, 2009.
[12] A. Bibal and B. Frénay. Interpretability of machine learning models and representations: an introduction. In *Proc. ESANN*, pages 77–82, 2016.
[13] A. Biere and A. Fröhlich. Evaluating CDCL Restart Schemes. In *Pragmatics of SAT*, 2015.
[14] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185. IOS Press, 2009.
[15] J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in isabelle/hol. In *International Symposium on Frontiers of Combining Systems*, pages 12–27, 2011.
[16] J. C. Blanchette, D. Greenaway, C. Kaliszyk, D. Kühlwein, and J. Urban. A Learning-Based Fact Selector for Isabelle/HOL. *Journal of Automated Reasoning*, 57(3):219–244, Oct 2016.
[17] D. Bojarski, Mariusz anf Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to end learning for self-driving cars. Available at arXiv:1604.07316, 2016.
[18] M. Bojarski, P. Yeres, A. Choromanaska, K. Choromanski, B. Firner, L. D. Jackel, and U. Muller. Explaining how a deep neural network trained with end-to-end learning steers a car. Available at arXiv:1704.07911, 2017.
[19] J. Bridge. *Machine Learning and Automated Theorem Proving*. PhD thesis, University of Cambridge, 2010.
[20] J. Bridge, S. Holden, and L. Paulson. Machine Learning for First-Order Theorem Proving – Learning to Select a Good Heuristic. *Journal of Automated Reasoning*, 53(2):141–172, Aug 2014.
[21] S. Chaki, E. Clarke, N. Sinha, and P. Thati. Automated assume-guarantee reasoning for simulation conformance. In *CAV*, pages 534–547, 2005.
[22] T. Chappelly, C. Cifuentes, P. Krishnan, and S. Gevay. Machine learning for finding bugs: An initial report. In *IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation*, 2017.
[23] E. Clarke, A. Gupta, and O. Strichman. SAT-Based Counterexample Guided Abstraction Refinement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(7):1113–1123, 2004.
[24] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
[25] E. M. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT-Based Abstraction-Refinement Using ILP and Machine Learning Techniques. In *Computer Aided Verification*, pages 265–279, 2002.
[26] E. M. Clarke and J. M. e. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.
[27] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, pages 331–346, 2003.
[28] P. Cousot and R. Cousot. A Gentle Introduction to Formal Verification of Computer Systems by Abstract Interpretation. In *Logics and Languages for Reliability and Security*, pages 1–29. IOS Press, 2010.
[29] D. Devlin and B. O'Sullivan. Satisfiability as a classification problem. In *Irish Conference on Artificial Intelligence and Cognitive Science*, 2008.
[30] H. Duncan, A. Bundy, J. Levine, A. Storkey, and M. Pollet. The use of data-mining for the automatic formation of tactics. In *Proceedings of Computer-Supported Mathematical Theory Development*, 2004.
[31] A. Elizabeth, M. Alan, B. H. Duncan, and D. Aspinall. PG Tips: A Recommender System for an Interactive Theorem Prover, 2006.
[32] E. Frank, M. A. Hall, and I. H. Witten. *The WEKA Workbench*. Morgan Kaufmann, 2016.
[33] A. Frohlich, A. Biere, C. Wintersteiger, and Y. A. . Hamadi. Stochastic local searchf or satisfiability modulo theories. In *AAAI*, pages 1136–1143, 2015.
[34] M. Fuchs. A Feature-based Learning Method for Theorem Proving. *American Association for Artificial Intelligence*, 1998.
[35] M. Gagliolo and J. Schmidhuber. Learning restart strategies. In *International Joint Conference on Artifical intelligence*, 2007.
[36] M. Gheorghiu, D. Giannakopoulou, and C. S. Pasareanu. Refining interface alphabets for compositional verification. In *TACAS*, pages 292–307, 2007.
[37] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.
[38] A. Grabowski, A. Kornilowicz, and A. Naumowicz. Mizar in a nutshell. *Journal of Formalized Reasoning*, 3(2):153–245, 2010.
[39] S. Haim and T. Walsh. Restart strategy selection using machine learning techniques. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 312–325, 2009.
[40] Q. Hanam, L. Tan, R. Holmes, and P. Lam. Finding patterns in static analysis alerts: Improving actionable alert ranking. In *Mining Software Repositories*, pages 152–161, 2014.
[41] H. Hata, O. Mizuno, and T. Kikuno. An extension of fault-prone filtering using precise training and a dynamic threshold. In *Mining Software Repositories*, pages 89–98, 2008.
[42] S. Heckman and L. Williams. A model building process for identifying actionable static analysis alerts. In *ICST*, pages 161–170, 2009.
[43] S. Heckman and L. Williams. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, 53:363–387, 2011.
[44] E. Horvitz, Y. Ruan, C. Gomes, H. Kautz, B. Selman, and M. Chickering. A bayesian approach to tackling hard computational problems. In *Conference on Uncertainty and Artificial Intelligence*, pages 235–244, 2001.
[45] J. Huang. The effect of restarts on the efficiency of clause learning. In *International Joint Conference on Artifical intelligence*, pages 2318–2323, 2007.
[46] F. Hutter, Y. Hamadi, H. H. Hoos, and K. Leyton-Brown. Performance prediction and automated tuning of randomized and parametric algorithms. In *Conference on Principles and Practice of Constraint Programming*, 2006.
[47] C. Kaliszyk and J. Urban. Learning-assisted automated reasoning with flyspeck. *Journal of Automated Reasoning*, 53(2):173–213, Aug 2014.
[48] C. Kaliszyk, J. Urban, and J. Vyskocil. Machine learner for automated reasoning 0.4 and 0.5. *CoRR*, abs/1402.2359, 2014.
[49] C. Kaliszyk, J. Urban, and J. Vyskočil. Efficient semantic features for automated reasoning over large theories. In *International Conference on Artificial Intelligence*, pages 3084–3090, 2015.
[50] H. Kautz, E. Horvitz, Y. Ruan, B. Selman, and C. Gomes. Dynamic randomized restarts: Optimal restart policies with observation. In *American Association for Artificial Intelligence*, 2002.
[51] S. Kim, E. J. J. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.
[52] B. Kitchenham. Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical report, University of Durheim and Keele University, 2007.
[53] E. Komendantskaya, J. Heras, and G. Grov. Machine learning in proof general: Interfacing interfaces. In *International Workshop On User Interfaces for Theorem Provers*, pages 15–41, 2012.
[54] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler. Correlation exploitation in error ranking. In *Foundations of Software Engineering*, 2004.
[55] D. Kühlwein, J. C. Blanchette, C. Kaliszyk, and J. Urban. MaSh: Machine Learning for Sledgehammer. In *Proceedings of the 4th International Conference on Interactive Theorem Proving*, ITP'13, pages 35–50, Berlin, Heidelberg, 2013. Springer-Verlag.
[56] D. Kühlwein and J. Urban. Males: A framework for automatic tuning of automated theorem provers. *Journal of Automated Reasoning*, 55(2):91–116, 2015.
[57] D. Kühlwein, T. van Laarhoven, E. Tsivtsivadze, J. Urban, and T. Heskes. Overview and evaluation of premise selection techniques for large theory mathematics. In *International Joint Conference on Automated Reasoning*, pages 378–392, 2012.
[58] M. G. Lagoudakis and M. L. Littman. Learning to select branching rules in the dpll procedure for satisfiability. *Electronic Notes in Discrete Mathematics*, 9:344–359, 2001.
[59] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. A portfolio approach to algorithm selection. In *IJCAI*, pages 1542–1563, 2003.
[60] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Exponential recency weighted average branching heuristic for SAT solvers. In *ICAI*, 2016.
[61] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Learning rate based branching heuristic for SAT solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 123–140, 2016.
[62] J. H. Liang, H. Govind, P. Poupart, K. Czarnecki, and V. Ganesh. An empirical study of branching heuristics through the lens of global learning rate. In *Theory and Applications of Satisfiability Testing (SAT)*, 2017.
[63] M. Lindauer, H. H. Hoos, F. Hutter, and T. Schaub. Autofolio: An automatically configured algorithm selector. *JAIR*, 53(1):745–778, 2015.
[64] S. M. Loos, G. Irving, C. Szegedy, and C. Kaliszyk. Deep network guided proof search. *CoRR*, abs/1701.06972, 2017.
[65] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Architectural Support for Programming Languages and Operating Systems*, pages 329–339, 2008.
[66] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of las vegas algorithms. In *Israeli Symposium on the Theory and Computing Systems*, 1993.
[67] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
[68] mboxThe Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2009.
[69] J. Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, 7(1):41 – 57, 2009.

[70] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Conference on Artificial Intelligence*, pages 1287–1293, 2016.
[71] T. Muske and A. Serebrenik. Survey of approaches for handling static analysis alarms. In *Source Code Analysis and Manipulation*, 2016.
[72] W. Nam and R. Alur. Learning-based symbolic assume-guarantee reasoning with automatic decomposition. In *ATVA*, pages 170–185, 2006.
[73] H. V. Nath and B. M. Mehtre. Static malware analysis using machine learning methods. In *International Conference on Security in Computer Networks and Distributed Systems*, pages 440–450, 2014.
[74] S. Nejati, J. H. Liang, V. Ganesh, C. H. Gebotys, and K. Czarnecki. Adaptive restart and cegar-based solver for inverting cryptographic hash functions. In *Verified Software: Theories, Tools, and Experiments*, pages 120–131, 2017.
[75] C. S. Pasareanu, D. Giannakopoulou, M. Gheorghiu Bobaru, J. M. Cobleigh, and H. Barringer. Learning to divide-and-conquer: Applying the l^* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 32(3):175–205, 2008.
[76] E. Pira, V. Rafe, and A. Nikanjam. Emcdm: Efficient model checking by data mining for verification of complex software systems specified through architectural styles. *Applied Soft Computing*, 49:1185–1201, 2016.
[77] M. R. Prasad, A. Biere, and A. Gupta. A Survey of Recent Advances in SAT-Based Formal Verification. *STTT Journal*, 7(2):156–173, 2005.
[78] J.-P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *International Symposium on Programming*, pages 337–351, 1982.
[79] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.
[80] T. Rocktäschel and S. Riedel. End-to-end differentiable proving. *Neural Information Processing Systems*, 2017.
[81] Y. Ruan, E. Horvitz, and H. Kautz. Restart policies with dependence among runs: A dynamic programming approach. In *Conference on Principles and Practice of Constraint Programming*, pages 573–586, 2002.
[82] N. Rungta and E. G. Mercer. Hardness for explicit state software model checking benchmarks. In *Software Engineering and Formal Methods*, pages 247–256, 2007.
[83] J. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionablestatic analysis warnings: An experimental approach. In *International Conference on Software Engineering*, pages 341–350, 2008.
[84] J. Sahs and L. Khan. A machine learning approach to android malware detection. In *Intelligence and Security Informatics Conference*, pages 141–147, 2012.
[85] H. Samulowitz and R. Memisevic. Learning to solve qantified boolean formulas. In *ICAI*, pages 255–260, 2007.
[86] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.
[87] D. Selsam, M. Lamm, B. Bunz, P. Liang, L. de Moura, and D. Dill. Learning a sat solver from single-bit supervision. Technical report, Stanford University, 2018.
[88] S. Shivaji, J. Whitehead, R. Akella, and S. Kim. Reducing features to improve code change based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569, 2013.
[89] D. Silver, A. Huang, C. J. Maddison, A. Guez, and L. e. a. Sifre. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.
[90] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, and M. e. a. Lai. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv:1712.01815*, 2017.
[91] N. Sinha and E. M. Clarke. Sat-based compositional verification using lazy learning. In *Conference on Computer Aided Verification*, pages 39–54, 2007.
[92] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction.* MIT Press, 1998.
[93] J. Urban. MizarMode – An Integrated Proof Assistance Tool for the Mizar way of Formalizing Mathematics. *Journal of Applied Logic*, 4(4):414 – 427, 2006.
[94] D. S. Wei Ting, C. Y.-L. Cheung, and G. Lim. Development and validation of a deep learning system for diabetic retinopathy and related eye diseases using retinal images from multiethnic populations with diabetes. *Journal of American Medical Association*, 318(22):2211–2223, 2017.
[95] C. Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 38:1–38:10, 2014.
[96] D. Wolpert and W. MacReady. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.
[97] H. Wu. Improving SAT solving with machine learning. In *ACM SIGCSE Technical Symposium on Computer Science Education*, pages 787–788, 2017.
[98] L. Xu, H. H. Hoos, and K. Leyton-Brown. Hierarchical hardness models for SAT. In *Conference on Principles and Practice of Constraint Programming*, 2007.
[99] L. Xu, H. H. Hoos, and K. Leyton-Brown. Predicting satisfiability at the phase transition. In *ICAI*, pages 584–590, 2012.
[100] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla: Portfolio-based algorithm selection for sat. *JAIR*, 32:565–606, 2008.
[101] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized Vulnerability Extrapolation using Abstract Syntax Trees. In *Annual Computer Security Applications Conference*, pages 359–368, 2012.
[102] S. Y. Yerima, S. Sezer, and I. Muttik. Analysis of bayesian classification-based approaches for android malware detection. *IET Information Security*, 8(1):25–36, 2014.
[103] S. Y. Yerima, S. Sezer, and I. Muttik. Android malware detection using parallel machine learning classifiers. In *Next Generation Mobile Apps, Services and Technologies*, 2014.
[104] S. Y. Yerima, S. Sezer, and I. Muttik. Android malware detection: An eigenspace analysis approach. In *Science and Information Conference*, pages 1236–1242, 2015.