Prioritizing Runtime Verification Violations

Breno Miranda[‡] Igor Lima[‡] Owolabi Legunsen[†] Marcelo d'Amorim[‡]
[‡]Federal University of Pernambuco, Brazil
[†]University of Illinois at Urbana-Champaign, USA
{bafm, isol2}@cin.ufpe.br legunse2@illinois.edu damorim@cin.ufpe.br

***Abstract*—Runtime Verification (RV) can help find software bugs by monitoring formally specified properties during testing. A key problem when using RV during testing is how to reduce the manual inspection effort for checking whether property violations are true bugs. To date, there was no automated approach for determining the likelihood that property violations were true bugs to reduce tedious and time-consuming manual inspection.**

We present RVPRIO, the first automated approach for prioritizing RV violations in order of likelihood of being true bugs. RVPRIO uses machine learning classifiers to prioritize violations. For training, we used a labeled dataset of 1,170 violations from 110 projects. On that dataset, (1) RVPRIO reached 90% of the effectiveness of a theoretically optimal prioritizer that ranks all true bugs at the top of the ranked list, and (2) 88.1% of true bugs were in the top 25% of RVPRIO-ranked violations; 32.7% of true bugs were in the top 10%. RVPRIO was also effective when we applied it to new unlabeled violations, from which we found previously unknown bugs—29 bugs in 7 projects and two bugs in two properties. Our dataset is publicly available online.

I. INTRODUCTION

Runtime Verification (RV) [15] helps to find bugs that occur when behavioral properties[1] are violated. In RV, properties are expressed in formalisms amenable to efficient monitoring. Those properties are instrumented into the code, and monitored during program execution. If an execution does not satisfy a property, an *RV violation* (i.e., violation) is generated so that developers can check whether there is a true bug. RV improved a lot in recent years [11], [24], [43], [68], [74], [112], to the point where there are now proposals for using RV to find bugs during everyday software development and testing [59]–[63]. RV tools like JavaMOP [40] allow to express parametric properties [57], use different formalisms to express those properties, and dynamically monitor these properties efficiently, e.g., to monitor multiple properties in one test run.

Legunsen et al. [60], [62], [63] recently showed that performing RV during test executions is scalable and helped find hundreds of bugs that existing tests written by developers did not find. Legunsen et al. found these bugs from their large-scale empirical study [62], [63] in which they used JavaMOP to monitor test executions in hundreds of open-source projects, using properties of parts of the standard Java library API [58], [82]. Unfortunately, they also reported that the developer effort for inspecting violations is very high—RV during testing produces very many violations that developers need to manually inspect and there are various potential causes for a violation. A property violation could be indicative of a problem in the property, a bug in the RV tool, or a bug in a project's code and tests, i.e., a true bug.

To date, the only way for developers to check whether a violation indicates a true bug is through tedious and time-consuming manual inspection. To put the problem in perspective, Legunsen et al. [63] reported spending 1,200 hours to inspect, discuss and patch 852 violations. It took us over 9 hours to inspect 90 violations (Section V). These times are for violations obtained from only one program version. The problem of high developer manual inspection effort is (1) worsened as software evolves rapidly and (2) exacerbated by the fact that, currently, a vast majority of violations (roughly 1,271 of 1,495 in Legunsen et al.'s studies [62], [63]) are due to problems with existing properties, and are not true bugs.

Although RV was shown useful for finding bugs during testing, there is no automated approach for reducing developer's manual inspection effort by determining the likelihood that violations are true bugs. Automated approaches for reducing the manual inspection effort will make RV easier to use.

This paper proposes RVPRIO, the first automated approach for reducing developer manual inspection effort by prioritizing violations in order of likelihood of being true bugs. Currently, RVPRIO uses machine-learning classifiers to prioritize violations. We trained RVPRIO classifiers with 1,170 violations from Legunsen et al.'s studies [62], [63]. Each violation was previously labeled as being a true bug or not. The classifier computes the probability that a violation is a true bug, which RVPRIO then uses to prioritize new violations. A key benefit of RVPRIO is that developers can set a time budget for inspection and still inspect the most likely true bugs. Machine learning was previously applied to various software engineering problems, including other prioritization tasks (Section VII). To the best of our knowledge, RVPRIO is the first approach to use machine learning to assist developers to use RV during testing.

RVPRIO was effective. On the 1,170 labeled RV violations in the training dataset, RVPRIO had ∼90% effectiveness, compared with a theoretically optimal prioritizer that ranks all true bugs at the top. RVPRIO prioritizes 88.1% of true bugs in the top 25% and 32.7% of the true bugs in the top 10%. By prioritizing 88.1% of the true bugs in the top 25%, RVPRIO could have saved the time for inspecting 75% of the violations at the expense of missing 11.9% of true bugs. Finally, we conducted a case study in which we used RVPRIO to prioritize 278 new unlabeled violations from 11 Apache projects. We manually inspected the top 90 RVPRIO-ranked violations.

[1]We follow the definition of behavioral properties by Robillard et al. [93]—"a way to use an API as asserted by the developer or analyst, and which encodes information about the behavior of a program when an API is used".

```
1  Collections_SynchronizedCollection(Collection c, Iterator i) {
2   Collection c;
3   creation event sync after() returning(Collection c):
4    call(* Collections.synchronizedCollection(Collection)){ this.c = c;}
5   event syncMk after(Collection c) returning(Iterator i) :
6    call(* Collection+.iterator()) && target(c) && Thread.holdsLock(c){}
7   event asyncMk after(Collection c) returning(Iterator i):
8    call(* Collection+.iterator())&& target(c)&& !Thread.holdsLock(c){}
9   event access before(Iterator i) :
10   call(* Iterator.*(..)) && target(i) && !Thread.holdsLock(this.c){}
11  ere: (sync asyncMk) | (sync syncMk access)
12  @match{ RVMLogging.out.println(/*violation message*/); }}
```

(a) Collections_SynchronizedCollection (CSC) property

```
1  public class CharSet ... {
2   ...
3   set = Collections.synchronizedSet(...);
4   public boolean contains(final char ch) {
5  + synchronized(set) {
6      for (final CharRange range : set) {
7      if (range.contains(ch)) {
8       return true;
9      }
10    }
11 + }
12    return false; }}
```

(b) Commons Lang Bug found from CSC violation

Fig. 1: A JavaMOP property, CSC, and a bug that RV of CSC helped find

RVPRIO was also effective for prioritizing previously unseen violations. We found 29 previously unknown bugs in these projects, and two previously unknown bugs in two properties.

This paper makes the following contributions:

- ⋆ **Approach.** RVPRIO is the first automated approach for determining the likelihood that violations are true bugs, as way to reduce developer manual inspection effort.
- ⋆ **Evaluation.** We evaluate RVPRIO on the largest publicly-available labeled dataset of violations. RVPRIO ranked 88.1% of true bugs in the top 25% of violations.
- ⋆ **Case Study.** In our case study, RVPRIO reduced developer effort in inspecting RV violations and more efficiently found previously unknown bugs in projects and properties.
- ⋆ **Data.** We made all our data publicly available, including the newly-labeled violations from our case study.

II. Finding Bugs with RV

This section shows how RV helps find bugs during testing, and describe the inputs and outputs of an RV tool. It exemplifies how RV works and the cost of inspecting violations.

RV Tool Input, Process and Output. Inputs to an RV tool are formally specified properties, a program and a way to run the program, e.g., tests. An RV tool instruments the properties into program. While executing the instrumented program, relevant *events* (e.g., method calls or field accesses) are generated and objects called *monitors* are created to check whether events satisfy the properties. The RV tool outputs violations which inform the tool's users that some properties were not satisfied

The RV tool that we used. All violations used in this paper came from an RV tool called JavaMOP [40]. Our choice is pragmatic—the violations that we use for training classifiers were generated from Java programs, JavaMOP is quite robust, easy to integrate with testing frameworks, publicly available, has been used in recent papers on performing RV during testing of open-source code [59], [60], [63], and is widely used in RV research, e.g., [37], [43], [71], [72], [85], [86]. So, to evaluate RVPRIO on previously unseen violations, we also used JavaMOP to monitor the same properties from previous work on a new set of Java projects (Section V).

Properties and Violations. Figure 1a shows a property, `Collections_SynchronizedCollection` (henceforth called `CSC`), in JavaMOP syntax [41]. Properties have three parts: (1) *event definitions* which specify relevant runtime method calls or field accesses, (2) *a specification*, which is a logical formula over the events, and (3) *a handler*, which is code that is run when events violate/match the specification.

`CSC` defines four events (lines 3–10). The `sync` event (lines 3–4) is triggered after calling `Collections.synchronizedCollection` to create a synchronized `Collection c`. The `syncMk` event (lines 5–6) is triggered after calling `Collection.iterator` to get an iterator, `i`, of `c` in a thread that locks `c`. In contrast, `asyncMk` (lines 7–8) is triggered after calling `Collection.iterator` to get `i` from `c` in a thread that does not lock `c`. Lastly, event `access` (lines 9–10) is triggered before accessing `i` in a thread that does not lock `c`.

`CSC`'s specification is defined on line 11 as an Extended Regular Expression (ERE) which matches if either (1) `i` is created from `c` without locking `c`, or (2) `i` is created from `c` after locking `c` but then `i` is subsequently accessed in a thread that does not lock `c`. In addition to EREs, JavaMOP supports several logical formalisms for expressing property specifications, e.g., FSM, CFG, LTL. The plugin-based design of JavaMOP allows one to easily add other formalisms.

If the ERE on line 11 is matched at runtime, then the code in the handler on line 12 is triggered. Handler code can be anything the RV tool user wants, such as failure recovery code. For our experiments, the handler simply prints a violation that warns users that `CSC` was violated and the program may have a bug.

```
Specification
 Collections_SynchronizedCollection has
 been violated on line CharSet.contains(
 CharSet.java:6)
 [17]. A synchronized collection was
 accessed in a thread−unsafe manner.
```

Fig. 2: An example property violation

Figure 2 is an example violation. Violations have four parts: the property name, the line of code where the last event that violated the property's specification occurred, a URL to the property definition (not shown in Figure 2), and a brief description. Since the same code can be executed multiple times during test execution (e.g., code in a loop or code covered by multiple tests), we only consider in this paper the set of unique violations. We consider two violations of the same property that occur at the same location to be the same.

Properties used in this paper. We used parametric multi-object properties [57] that were manually formalized from the Javadoc of several Java APIs. Specifically, Lee et al. [58], [68] read through the Javadoc of a subset of four Java packages (`java.lang`, `java.io`, `java.util`, and `java.net`). They manually identified and formalized, as properties, sentences that describe "must", "should" or "is better to" conditions,

using the aforementioned formalisms as appropriate. Lee et al. assigned one of three *severity levels* to each property—(1) `error`: violations of the property are expected to always indicate bugs, (2) `warning`: violations of the property may be bugs in some scenarios but not in others, and (3) `suggestion`: violations of the property merely indicate bad programming practice. Legunsen et al. [62], [63] found both true bugs and false alarms from inspecting violations of properties in all three severity levels. So, we used these severity levels as features learned by our classifiers (Section III-A). All 182 properties by Lee et al. are publicly available [84], and we used all but the 21 that Legunsen et al. [63] found defective.

Finding bugs from RV during testing of open-source code. To see how RV of `CSC` helps find bugs, consider the buggy and fixed version of the code snippet in Figure 1b. The code shows a true bug that RV of CSC helped us find in Apache Commons Lang [4], a widely-used library of utilities for manipulating core classes in the standard Java library. In Figure 1b, lines that do not begin with "+" represent the buggy code, and lines that start with "+" represent the fix. In the buggy version, line 3 creates a synchronized `Collection`, `set`, triggering `CSC`'s `sync` event. Line 6 creates a `set` iterator without locking `set`, i.e., `CSC`'s `asyncMk` event. This event sequence from the buggy version matches `CSC`'s specification, so JavaMOP generates the violation in Figure 2. Our inspection, prompted by the violation, confirmed that multiple threads can access line 6 in Figure 1b and lead to bugs due to non-determinism. We made a pull request for the fix on lines 5–11, which was accepted [18].

Problems in dealing with violations in practice. Currently, developers face three major problems when dealing with violations from RV of test executions. First, very many violations are generated that developers need to inspect. As examples, RV on one version of Apache Commons lang produced 61 violations and Legunsen et al. [60] found 643 violations from 10 open-source projects. Second, inspecting a violation can be very time consuming, especially if events leading to the violation happen in parts of code that a developer is not familiar with or are due to interactions with third-party libraries. Legunsen et al. [63] estimated that it took 1,200 hours to inspect 852 violations. Third, as many as 85.9% of violations obtained from monitoring existing properties are false alarms—they do not help find true bugs in the code under test [62], [63]. To illustrate, `CSC` is violated even if the code in Figure 1b only runs in a single-threaded context; a bad programming practice that cannot lead to bugs due to non-determinism. In fact, `XStream` developers rejected Legunsen et al.'s pull request fixing a CSC violation because they argued that their code is not intended to be thread safe.

Researchers [62], [63], [102] have argued that better properties will lead to fewer false alarms. But, developers who want to use RV to find bugs today (because RV helped find many bugs) will have to deal with false alarms despite tremendous research progress on mining properties [7], [14], [19], [26], [54], [56]–[58], [64], [75], [76], [81]–[83], [91], [93], [109], [113], [118]. Therefore, our goal in this paper is to reduce developer manual effort for inspecting violations.

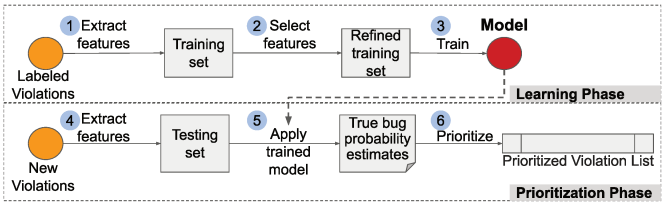

Fig. 3: Overview of RVPRIO's classification-based prioritization.

III. Approach

RVPRIO uses binary classifiers to compute probabilities of a violation being a true bug (respectively, false alarm), and then uses that information to prioritize previously unseen violations. Machine Learning was used before for other prioritization tasks in software engineering, e.g., ranking alarms from static analysis tools [36], [50]. To the best of our knowledge, this is the first approach for prioritizing violations from an RV tool; RV was successfully used to find many bugs [62], [63].

RVPRIO takes a set of violations as input and produces a ranking of those violations as output. Figure 3 illustrates the workflow of RVPRIO. The first layer illustrates the learning (offline) phase that builds a prediction model based on a training dataset of previously-labeled violations. The model computes the probability that a violation is a true bug. The second layer shows the prioritization (online) phase that ranks unlabeled violations. The following sections describe our feature extraction process and the RVPRIO phases.

A. *Features used in* RVPRIO

Selecting informative, discriminating, and independent features is an important step when solving a classification problem. Conceptually, learning methods infer combinations of features capable of *predicting* the class of an object under observation. In our context, the object under observation is the violation. In RVPRIO, features are characteristics of violated properties and the code that triggered those violations.

Feature Extraction. We extracted features 1) *from the property* associated with the violation and 2) *from the source code* that triggered the violation. From the property, we considered the name of the violated property and its severity. From the source code, we considered various metrics related to internal code quality, obtained using PMD [78]. PMD is a static analyzer that provides several built-in rules, organized in different categories. We used the category *Error Prone*, which is related to functional behavior. We also used standard metrics of code quality (e.g., number of lines of code, cyclomatic complexity, number of tokens, and number of function parameters). Note that static code attributes were used to solve other classification problems [28], [53], [70], [87].

Feature Selection. Non-informative and non-discriminating features can negatively affect prediction quality [20]. Therefore, we applied a standard method for finding an optimal selection of features known as Recursive Feature Elimination with Cross-Validation (RFECV) [29]. In RFECV, first the importance of each feature is obtained by training an estimator on the original set of features. Then, the least

important features are recursively excluded and the model is re-trained. If model performance becomes worse, the RFECV process stops and any remaining features are selected. Note that some of the classification algorithms that we used can automatically ignore irrelevant features, i.e., they internally perform feature selection. Nevertheless, Chandrashekar and Sahin [13] showed that even these algorithms can benefit from independent feature selection. After applying RFECV, we reduced the original set of 110 features to 47 features. The three most important features according to our model were: 1) TOKEN, 2) SPEC_is_ByteArray..._FlushBeforeRetrieve, and 3) NLOC. TOKEN denotes the number of tokens in the function that triggered the violation, the second feature refers to the name of a property frequently violated in the dataset, and NLOC is the number of lines of code of function. The complete list of features selected and their importance can be found online: https://github.com/brenomiranda/rvprio.

B. Learning Phase

Figure 3 shows each processing step in RVPRIO's workflow with a numbered arrow. The learning phase starts with feature extraction (Step 1), which takes a set of labeled violations as input. For each violation, we extract features from the violated property and from the code that triggered that violation. The output of this step is the raw training set, i.e., a table where rows are violations and columns, except the "class" column (indicating whether the violation is true bug or false alarm), are features of that violation. Then, feature selection (Step 2) eliminates uninformative (i.e., not selected) features from the input training set. The refined training set is then used for training the model, which is the output of the last step (Step 3) of the learning phase.

C. Prioritization Phase

In Step 4, RVPRIO processes new violation to produce a testing set with the same features selected during the learning phase. In Step 5, RVPRIO takes the testing set, applies the learned model to predict the probability that each violation is a true bug and outputs the list of all violations with their respective estimated probabilities of being a true bug. Finally, in Step 6, the violations are ranked by their predicted probability of being a true bug, from highest to lowest.

IV. Evaluation

We pose the following research questions.

- **RQ1.** How good are different binary classifiers for predicting whether violations are true bugs?
- **RQ2.** How good is RVPRIO for prioritizing violations?

The first question measures the performance of classifiers whereas the second question evaluates the performance of RVPRIO, which is influenced—but not determined—by the probabilities that a binary classifier assigns to labels.

Dataset. To train classifiers, we used the dataset from Legunsen et al. [62], [63]. That dataset includes 1,495 *labeled* violations. However, 200 violations are from automatically mined properties from which it is not possible to extract

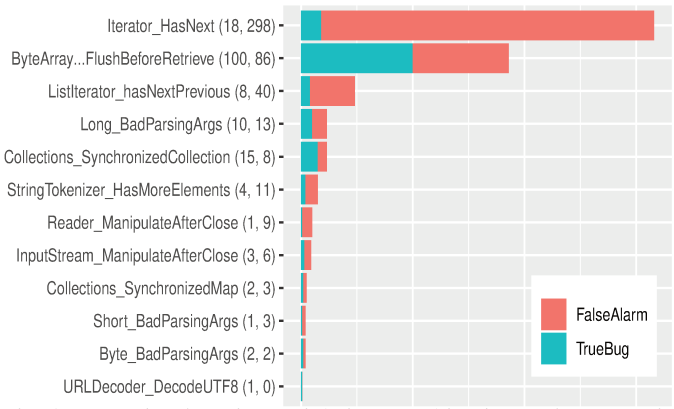

Fig. 4: Properties that trigger violations resulting in true bugs. Y-axis is labeled as "Property(#TrueBugs, #FalseAlarm)".

all features required by our model (e.g., the severity level of the property). Also, some projects used to obtain the original dataset are no longer available on GitHub, so we could not obtain their source code. After removing violations of automatically mined properties and unavailable projects, our final training dataset contained 1,170 labeled violations from 110 projects and 43 properties. 12 of these 43 properties produced 165 violations that were labeled as true bugs, i.e., 14.1% of the training dataset were true bugs. Figure 4 shows these 12 properties and the number of true bugs and false alarms from inspecting their violations. The x-axis shows the total number of violations, whereas the y-axis shows the name of the violated properties and, inside parentheses, the number of true bugs and false alarms produced, respectively.

A. Classifiers

RVPRIO prioritizes violations using the probability of each violation being a true bug, as computed by off-the-shelf binary classifiers. We evaluated RVPRIO using a variety of classifiers from Python's *scikit-learn* library [77], using representatives of the main types of algorithms—probabilistic, linear classifiers, nearest neighbor, decision trees, and ensembles [23], [111]. Column "Classifier" in Table I lists the classifiers we used, including a random classifier, dubbed DummyClassifier, which serves as the baseline to measure how other classifiers compare with random classification—it generates random predictions with respect to the distribution of labels in the training set. The DummyClassifier is also available on scikit-learn.

B. Metrics and Methodology

We used standard metrics to evaluate the performance of binary classifiers and the corresponding prioritizer.

1) RQ1: To evaluate the performance of classifiers we used Precision (Pr), Recall (Re), and the F1-score ($F1$) [101]. Precision is a proxy for measuring the amount of false positives (i.e., false alarms incorrectly classified as true bugs), recall is a proxy for measuring the amount of false negatives (i.e., true bugs incorrectly classified as false alarms), and the F1-score averages precision and recall. In all cases, the higher the value the better. Note that it is important to

TABLE I: Performance indicators of classification algorithms obtained with 30 repetitions of 10-fold cross validation.

Classifier	Precision	Recall	F1-score
Gradient Boosting Classifier	0.74	0.73	0.73
Logistic Regression	0.76	0.70	0.71
Random Forest	0.73	0.70	0.71
AdaBoost	0.74	0.68	0.70
Decision Tree	0.70	0.68	0.69
Gaussian Processes	0.67	0.65	0.66
Neural Net	0.74	0.65	0.64
Nearest Neighbors	0.67	0.63	0.64
Naive Bayes	0.62	0.75	0.55
Linear SVM	0.70	0.54	0.53
DummyClassifier	0.50	0.50	0.50

balance the two metrics—100% precision with 0% recall can be obtained by not classifying any violation as a true bug and 0% precision with 100% recall can be obtained by classifying all violations as true bugs. Neither option is useful. The following equations define these metrics, where TP, FP, TN, and FN denote, respectively, the number of true positives, false positives, true negatives, and false negatives: $Pr = TP/(TP + FP)$, $Re = TP/(TP + FN)$, and $F1 = 2 * (Pr * Re)/(Pr + Re)$.

We used k-fold (with k=10) cross validation [38] to evaluate classifier performance. This evaluation method randomly divides the dataset into k groups, or folds, of approximately equal sizes. Then, for each fold, it treats the selected fold as the validation set and all remaining k-1 folds as the training set. After iterating through all folds, it averages the evaluation metrics. Because our dataset is imbalanced, i.e., the number of observations with "true bug" labels is significantly lower than those with "false alarm" labels, we used *stratified* cross validation [66] to guarantee that the folds preserve the percentage of samples in each class. To perform a more robust model assessment and to account for the random splitting of data into folds, we repeated cross validation 30 times, such that the folds are split differently each time. Table I shows the average across 30 repetitions of stratified 10-fold cross validation. Note that these are standard machine learning metrics and evaluation procedures. We did not (re-)implement them; we used the implementation in the *scikit-learn* library [77].

2) RQ2: To evaluate prioritization performance, we used a metric that was originally proposed by Rothermel et al. to evaluate performance of test case prioritization techniques [94]. We refer to our metric as Average Percentage of Bugs Detected (APBD)[2]. APBD computes the weighted average of the percentage of true bugs revealed over the list of violations. APBD values range from 0 to 1 (frequently, they are also reported as percentage) with high values indicating faster detection of "true bugs" in the list of violations. For a given set of violations, the optimal APBD is achieved when all "true bugs" appear are at the top of the list followed by all false alarms. The definition of APBD is: $\text{APBD} = 1 - \frac{VB_1+VB_2+...+VB_m}{nm} + \frac{1}{2n}$. Given a set of n violations V, and a set of m bugs B that can be revealed by analyzing V, for any possible ordering of V, we call VB_i the position in the ranked list of the first violation that reveals bug i. For example, consider the case where we have 3 violations (n=3) and only one of them reveals a true bug (m=1). If RVPRIO ranks the "true bug" violation at the top of the list, APBD is 1 - 1/3 + 1/6 = 5/6 (∼83%). If, on the other hand, RVPRIO ranks the "true bug" violation at the bottom of the list, APBD is 1 - 3/3 + 1/6 = 1/6 (∼17%).

[2]The original name of the metric is APFD, where "F" refers to Faults.

C. Answering RQ1: How good are different binary classifiers for predicting whether violations are true bugs?

Table I displays the average precision, recall, and F1-score for various classifiers. Techniques are sorted by column "F1-score", which is the most relevant metric for RVPRIO; it takes into account false positives *and* negatives. So, techniques with higher F1-score are considered better. DummyClassifier appears at the bottom of the table with an F1-score of 50%.

Table I shows that Gradient Boosting Classifier (GBC) achieved the best overall performance, with an F1-score of 0.73. Note that precision and recall are well balanced for this classifier. GBC is closely followed by Logistic Regression, Random Forest, and AdaBoost. Interestingly, GBC, Random Forest, and AdaBoost are ensemble techniques that use a portfolio of methods to compute the prediction model. GBC, in particular, is an ensemble that builds on weak prediction models, typically decision trees [25]. This result is consistent with recent work showing the benefit of using multiple classifiers in solving various classification problems [111]. Considering the bottom of the table, Naive Bayes and Linear SVM had F1-scores similar to DummyClassifier. To sum up:

> Binary classification was effective. Four of the algorithms analyzed produced an F1-score above or equal 70%. Three of these are ensembles.

GBC had the best overall performance. So, in the rest of this paper, we instantiate RVPRIO with GBC (i.e., RVPRIO-GBC).

D. Answering RQ2: How good is RVPRIO for prioritizing violations?

RQ2 evaluates how well RVPRIO prioritizes RV violations. Figure 5 shows the progress of true bugs revealed as violations are analyzed. The x-axis indicates the number of violations analyzed; the y-axis shows the percentage of true bugs revealed. The faster a curve reaches 100 on the y-axis the better.

These curves represent the cumulative percentage of true bugs revealed for different prioritization strategies. The area under the curve denotes the weighted average of the percentage of true bugs revealed over the analysis of the violations, i.e., the APBD of the prioritization. The dotted (blue) curve is the theoretically optimal prioritization order with all true bugs at the top of the list. Optimal order has an APBD of ∼94%. The other curves show the performance of RVPRIO using GBC and random prioritization. The solid (green) curve shows the progress of bugs revealed in RVPRIO-GBC-prioritized order. There is a sharp growth at the beginning, reaching a peak of

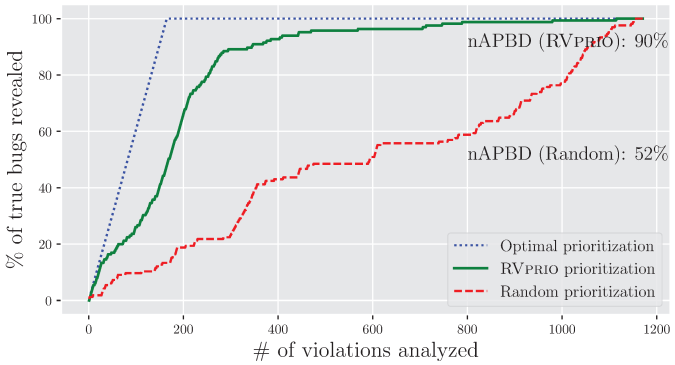

Fig. 5: Performance of different prioritization strategies.

82% of true bugs revealed after analyzing only 19.82% (268) of the violations. For comparison, with random prioritization, a developer would need to analyze 1,202 violations (89%) to obtain the same results. From that point on, the number of true bugs revealed by RVPRIO-GBC increases slowly, achieving a normalized APBD (nAPBD) of ∼90% at the end. We normalized the APBD values—highlighted in the plot—of RVPRIO-GBC and random prioritization to facilitate the comparison of the techniques with optimal prioritization. nAPBD is obtained by dividing the APBD of a technique by the optimal APBD (∼94%). Random prioritization, shown as a dashed (red) curve, achieves nAPBD of ∼52%.

TABLE II: Percentage of true bugs revealed for different fractions of the violations list.

% of violations	5%	10%	15%	25%	50%	75%
RVPRIO-GBC	17.26%	32.74%	60.71%	88.10%	96.43%	97.62%
Random	4.17%	5.36%	14.29%	24.40%	56.55%	64.29%
Improv.	4.13x	6.11	4.25	3.61	1.71	1.51

Table II shows the percentage of true bugs revealed at different cutoff points, i.e., for increasing fractions of the prioritized list of violations. It is noticeable that RVPRIO-GBC is effective for smaller fractions. This is an important result with practical implications: the prioritization increases the likelihood that the most important violations (the ones with higher probability of being true bugs) will be analyzed first. In a scenario when only some of the violations can be inspected, the available analysis time will be spent inspecting the most critical ones. For comparison, considering the first 15% of the violations, RVPRIO-GBC reveals, on average, at least 5 times more bugs than random prioritization. To sum up:

> Classification-based prioritization was effective. RVPRIO-GBC achieved ∼90% of the effectiveness obtained by the optimal prioritization. Furthermore, 88.10% of the true bugs could be revealed after analyzing only 25% of the violations.

V. Case Study

This section reports the results of a case study that we performed to evaluate the benefits of RVPRIO when inspecting new unlabeled violations. A student (co-author of this paper) inspected an RVPRIO-prioritized list of new RV violations. We describe the projects and violations that we used, the procedure that we followed, and time savings that could be obtained from inspecting a fraction of prioritized violations. Finally, we show a sample of the violations that we inspected.

Violations. The violations in this case study are new, unlabeled, and not used in any prior study. Recall that violations in the training dataset from Section IV already had "true bug"/"false alarm" labels assigned, which we used as ground truth to evaluate the classifier and RVPRIO performance. That experiment did *not* involve developer inspection beyond what Legunsen et al. [62], [63] had done to assign labels.

Projects. We analyzed Apache projects, which we expected to contain tests and to have developers likely interested in potential bug reports that we could submit. We started from a list of 137 Apache Java projects that run builds on the publicly-viewable Apache Jenkins server [3]. We first removed projects that were already used in the training dataset (Section IV) and then selected projects that compile with Java 8 or higher, and which use the Maven build system. This resulted in 66 projects. Among these, there were only 11 projects that we were able to build, test, and monitor with JavaMOP. We used only these 11 projects in our case study. Table III shows the projects and their versions (SHAs) that we used.

Procedure. First, we ran JavaMOP on, and collected violations from, these 11 projects using the properties described in Section II. Column "Violations" in Table III is the number of violations per project, "Inspected" is the number of violations per project that we inspected, and "TB" is the number of true bugs that we found. Second, we used RVPRIO to prioritize the violations reported by JavaMOP. Notice that the prioritized list contains violations from all 11 projects. This reflects what would happen in ecosystems where multiple projects are built and tested together, e.g., in monolithic repositories [39], [80]. The student did *not* inspect all 278 violations, but stopped after the first 90 ranked violations. Thus, the proportion of inspected violations is not uniform in Table III—some projects had no violation ranked in the top 90. The false alarm rate observed in prior studies [62], [63] was around 90%. So we stopped the inspection after identifying a number of true bugs that corresponds to nearly 10% of the 278 violations. We found 29 new true bugs—10.4% of the 278 total violations.

A. RVPRIO *Performance on Unlabeled Dataset*

Figure 6 shows the performance of techniques in terms of APBD (see Section IV-B2) for the inspected violations. The solid green line shows the performance of RVPRIO compared with the optimal prioritization order (dotted blue line) and a randomized inspection order, averaged over 30 runs (dashed red lines). The plots in Figure 6 are similar to those in Figure 5—axes description and interpretation are

TABLE III: Case Study Projects and Violations

Apache projects	SHA	Violations	Inspected	TB
santuario-java	3f4f5f40	66	24	10
activemq-artemis	e2d6d072	46	12	5
struts	570f8c3e	45	12	2
pdfbox	4c6428d7	31	9	3
juneau	0e0c0a0a	27	10	2
stanbol	2fcf471b	23	12	0
tika	86325105	18	3	0
ranger	f80ee0e7	3	0	0
ambari	b0596110	8	0	0
hive	333264b2	6	6	6
shiro	010e4567	5	2	1
total		278	**90**	**29**

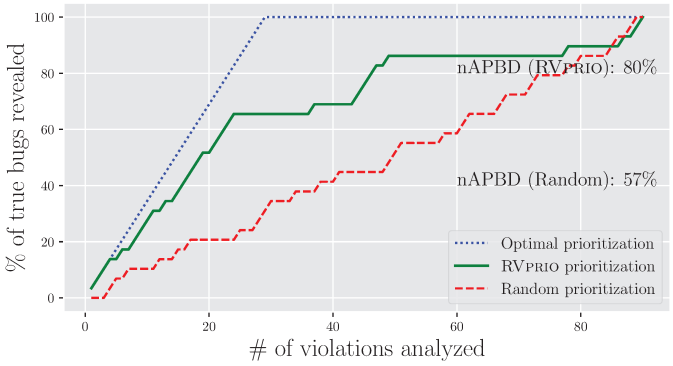

Fig. 6: Performance of RVPRIO on an unlabeled dataset

the same. RVPRIO prioritization resulted in nAPBD of 80%, while random order resulted in nAPBD of 57%.

Table IVa shows the percentage of true bugs revealed at different stages of the inspection process. Similar to our observation on the labeled dataset, these results seem to reinforce that RVPRIO provides greater benefit after a relatively small percentage of the violations have been inspected (up to 25%). Inspecting 5% of the violations in RVPRIO order revealed ∼14% of true bugs whereas random order revealed no bug at that cutoff point. (In practice, a developer using random order would likely have stopped inspecting randomly-ordered violations at this point [52]). Inspecting 10% of the violations in RVPRIO order revealed 24.14% of the true bugs vs. 3.45% bugs revealed with random order—a 6.9x difference. This difference reduces to 3.4x after inspecting 25% of the violations. The percentages of true bugs revealed at different stages in Table IVa seem lower than at corresponding stages in Table II, likely because we did not inspect all violations. Although Table IVa shows good but underestimated performance of RVPRIO, if we had inspected all violations, the ratio of false alarms to true bugs would likely have been higher and bring the results in Table IVa closer to those in Table II.

Kremenek and Engler [52] report four practical concerns that are important to developers when prioritizing source-code analysis (SCA) tool warnings; two may be relevant to RVPRIO: (1) the top three ranked warnings must not contain false alarms; otherwise a developer is likely to discard the tool and (2) after the top three, if the developer sees a sequence of 10–20 false alarms, they are likely to stop inspecting. For (1), the first four violations that RVPRIO ranked in our case study were true bugs. For (2), in our case study, stopping inspection

TABLE IV: Inspection progress.

% of violations	5%	10%	15%	25%	50%	75%
RVPRIO-GBC	13.79%	24.14%	34.48%	58.62%	72.41%	86.21%
Random	0.00%	3.45%	6.90%	16.24%	37.93%	65.52%
Improv.	n/a	6.99x	4.99	3.40	1.91	1.32

(a) True bug % revealed at different fractions of the list of violations.

% of true bugs	25%	50%	75%	100%
RVPRIO-GBC	111	175	292	543
Random	125	269	423	541

(b) Time required (in minutes) to reveal true bugs.

after the first 10 consecutive false alarms would have resulted in missing 9 of 29 true bugs. Stopping after the first 20 consecutive false alarms would have resulted in missing 3 of 29 true bugs. A developer who stops as Kremenek and Engler say would have done so before inspecting 90 violations—there was a sequence of 20 consecutive false alarms before the 90-violation mark in our case study.

During the case study, the co-author kept track of time spent inspecting each violation. It took 549 minutes to inspect all 90 violations—this includes neither time for a second co-author to double-check inspection results nor time to discuss and prepare pull requests. Average inspection time for true bugs was not much different from time for false alarms, as shown in the box plots in Figure 7. The average time to inspect true bugs was 6.4 mins (median=6); false alarms took 5.9 mins (median=4). In general, the amount of time required to inspect a violation is influenced by the inspection order. For example, when multiple violations of the same property, or from the same source file are clustered together (i.e., appear back-to-back in the prioritized list), developer's context switching is reduced and inspection times progressively get shorter. For illustration, if we assume that the time to inspect a violation is independent of inspection order, an optimal ordering (all true bugs at the top of the list) would have required 187 minutes to inspect all the true bugs, saving 362 minutes of developer time. Of course, in real world, it is impossible to know if all true bugs have been found until all violations are inspected. Hence we measured percentage of true bugs found at different stages (Table IVa).

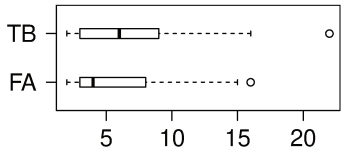

Fig. 7: Inspection time in mins (TB: true bugs, FA: false alarms)

Table IVb shows the time for the student to find 25%, 50%, 75%, and 100% of the true bugs in our case study. As a baseline for comparison, we also simulated how much time it would have taken to reveal the same number of true bugs if random order had been followed (averaged over 30 runs). For the first 50% of true bugs revealed, RVPRIO-GBC was at least twice as fast as the random order, but its rate of revealing true bugs is slower after the 50% point. This result matches our expectations. RVPRIO-GBC ranks all likely true bugs at the top of the list, followed by likely false alarms. When RVPRIO-GBC mis-classifies a violation, it is possible that human inspection will find a true bug

towards the very end of the prioritized order. This undesirable behavior reduces the performance of RVPRIO in the second half of Table IVb, and can be minimized only by further improving the classifier, i.e., training the model with more data, exploring other features, improving feature selection, etc. The performance of the random order in Table IVb is more stable; it eventually converges after averaging multiple repetitions, to a uniform distribution of true bugs.

B. Sample of Inspected Violations

This section discusses some violations that we inspected in our case study. Specifically, we discuss two true bugs and two false alarms as examples. It is worth noting that inspecting the violations for the two false alarms discussed helped us find a previously unknown bug in two JavaMOP properties.

1) Examples of True Bugs: **Apache Hive Bug.** Figure 8 shows code snippet for a true bug in Apache Hive. Lines in the original code do not start with "+"; line 7 is our fix, starting with a "+". The property violated is `ByteArrayOutputStream_FlushBeforeRetrieve` [12], or BAOS; it checks that an `OutputStream` built on a `ByteArrayOutputStream` is flushed or closed before calling `ByteArrayOutputStream.toByteArray()`, to avoid incomplete data in the resulting byte array. When testing the original code in Figure 8, BAOS is violated on line 8. The fix (line 7) flushes the `DataOutputStream` object to ensure that complete data is written from the `ByteArrayOutputStream`. Legunsen et al. [62], [63], found BAOS to be one of the more effective properties, with relatively low false alarm rates and dozens of BAOS-related bugs accepted by developers.

```
1 void doTestWriteReadFields(Random r, BigDecimal b)...{
2  HiveDecimal dec = HiveDecimal.create(b);
3  ByteArrayOutputStream baos = new ByteArrayOutputStream();
4  DataOutputStream out = new DataOutputStream(baos);
5  HiveDecimalWritable dwo = new HiveDecimalWritable(dec);
6  dwo.write(out);
7 +out.flush();
8  byte[] valueBytes = baos.toByteArray(); ... }
```

Fig. 8: True Bug found in Apache Hive.

Apache Juneau Bug. Figure 9 shows a code snippet for a true bug in Apache Juneau. Original code lines do not start with "+"; lines for our proposed fix start with "+". The property violated is CSC (described in Section II), which checks that code that iterates over a `synchronizedList`, `sl`, must do so from within a thread that locks `sl`, to avoid non-deterministic failures. The fix (lines 4 and 7) first locks the `listeners synchronizedList` before iterating over it. CSC helped find several confirmed bugs [60], [63].

```
1 listeners =
2   Collections.synchronizedList(new LinkedList<CL>());...
3 void onConfigChange(ConfigEvents events) {
4 +synchronized(listeners){
5   for (CL l : listeners)
6    l.onConfigChange(events);
7 +}
8 }
```

Fig. 9: True Bug found in Apache Juneau.

2) Examples of False Alarms: **Apache PDFBox.** Figure 10 shows code snippet for false alarms from `PDFBox`that led us to discover a previously unknown bug in two *properties*: `TreeSet_Comparable` [103] and `SortedSet_Comparable` [97]. These properties check that non-comparable elements are not added to a `SortedSet` or `TreeSet` via the respective `add()` or `addAll()` methods. An element is comparable if it implements the `Comparable` interface or uses a `Comparator`. However, in the event definition for `add()`, both properties were written to check for calls to `add*()`, which also matches calls to `addAll()`. For example, on line 4 in Figure 10, when the call to `addAll()` is made with a `Set` as argument, that call matches the erroneous event that is defined as `add*()`. Since `Set` objects are non-comparable, the properties are (wrongly) violated. We have reported this bug in both properties to the JavaMOP developers [98]. Interestingly, RVPRIO classified these violations as true bugs, with very high probability. Our conjecture is that the erroneous classification is influenced by the fact that both properties have severity level of `error`.

```
1 class TTFSubsetter { ...
2  SortedSet<Integer> glyphIds = new TreeSet<>(); ...
3  void addGlyphIds(Set<Integer> allGlyphIds) {
4   glyphIds.addAll(allGlyphIds);
5  }...
6 }
```

Fig. 10: Simplified Code for Apache PDFBox False Alarm

VI. Discussion

Lessons Learned. The main lesson we learned is that prioritization of RV violations is effective for significantly reducing developer manual effort for inspection. Regarding the use of machine learning in RVPRIO, our results show that features computed from *static* code and property characteristics are useful for prioritizing warnings from a *dynamic* analysis technique (RV in this case). Although we chose a machine-learning based approach in RVPRIO, we do not think that machine learning is the only way to approach the problem. RVPRIO merely opens a line of research on RV violation prioritization, and alternative approaches should be explored in the future. For example, it may be possible to develop program-analysis based techniques for prioritization and use them together with machine-learning based techniques like RVPRIO.

Our case study shows that it is important for future machine-learning based RV violation prioritization techniques to not just measure precision, recall and performance on labeled data sets, but to also evaluate how well they help developers who may be tasked with inspecting new, unlabeled violations. For example, had we not performed a case study we would not have been able to measure time savings that a developer can obtain. We would also not have experienced for ourselves the perspectives that potential users of RVPRIO may have. Unless individual software projects have many more violations than the ones in our case study, classifiers such as RVPRIO will likely become more accurate when there is more data available for training and testing. Thus, RVPRIO will likely be useful in

software ecosystems where many projects are tested together (and RV of those test executions is performed). Lastly, given the good results achieved by Logistic Regression (shown in Table I and discussed in Section IV-C), we expect it to perform similarly as GBC in practice, where Logistic Regression could be more attractive than ensembles due to its simplicity.

Limitations. As with any machine-learning based approach, RVPRIO evaluation is limited by the amount of labeled RV violations available for training and testing. We used all labeled RV violations that we could find from the literature. As more labeled violations become available, it will be interesting to see how the performance of RVPRIO improves. RVPRIO cannot rank violations of a newly-written property that it was not previously trained on. Such property would first have to be monitored in several projects and then its violations inspected, labeled and used for re-training RVPRIO's model.

Future Work. This paper opens up a new research direction on performing RV during software development. As recent research [60]–[63] showed that the overhead of performing RV during testing is becoming more tolerable, we believe that now is the time for more research on reducing manual developer effort for inspecting RV violations.

We highlight here some future work that we are interested in: (1) Investigating RVPRIO performance on project-specific properties, not just standard Java API properties, like we used in this paper. For example, the Apache Software Foundation has many projects whose APIs are well documented and which are widely-used as third-party libraries in other open-source projects. It may be possible to come up with properties from the API of these Apache projects, and investigate the effectiveness of RVPRIO for prioritizing violations that occur when running their tests and the tests of the projects that depend on them. (2) Enriching the features in RVPRIO classifiers. For example, in this paper, we only considered features from the method in which the last event occurred that led to a violation. Other features may be obtained from all methods in which all events occurred that led to each violation. Also, features may also be obtained from dynamic execution, e.g., how many times a violation occurred, or the number of tests during whose execution the same violation occurred. It would be interesting to see if and how much these additional features improve RVPRIO performance and accuracy. (3) We observed during our experiments that there may be room to make the violation inspection task even less burdensome for developers by combining prioritization with clustering. Violations of the same property often occur in code that have similar API usage patterns. (4) Incorporating information about software evolution into RVPRIO, both for obtaining more features for training and testing classifiers, and for utilizing decisions that developers made in the past to classify and prioritize future violations. In particular, it will be interesting to see how to use RVPRIO together with evolution-aware RV techniques [60].

Threats to Validity. *Construct validity.* To answer RQ1 and RQ2 we adopted standard metrics and techniques from the literature to evaluate classifiers and prioritizers. *Internal validity.* Developers can mis-classify violations. To mitigate this threat, we double checked every true bug reported. *External validity.* As usual, results may not generalize to other programs and properties. To mitigate this threat, we evaluated RVPRIO on a public dataset with 110 projects and 11 more projects in a case study. Lastly, the student who inspected the violations in our case study is a co-author on this paper, which may bias the results. To mitigate this, multiple co-authors rigorously double-checked the inspection process and results.

VII. Related Work

We describe related work on RV, machine learning in software engineering, and on using machine learning for dealing with warnings from static code analysis (SCA) tools.

A. Runtime Verification

The research on RV has been around for decades, and the kind of RV that we focus on in this paper—monitoring formally specified properties during program execution without requiring programmers to annotate their code—kicked off with seminal papers in the early 2000s [22], [32], [33]. Although it was previously recognized that RV can help find bugs during software development [15], [47], it was only recently that RV during software testing was feasible, thanks to the research and development of algorithms and techniques to make RV scale [1], [5], [6], [8]–[11], [21], [24], [31], [43]–[45], [68], [71], [72], [74], [85], [86], [112]. Specifically, Legunsen et al. recently started to explore the opportunities, benefits and challenges of performing RV during testing [59]–[63]. Although Legunsen et al. [62], [63] were the first to quantify the severity of the problem of false alarms that result from monitoring existing properties during software testing, RVPRIO is the first automated approach for determining the likelihood that violations are true bugs.

To reduce RV overhead, Legunsen et al. [59], [60] proposed to prioritize *properties* used in RV during software evolution. RVPRIO is the first to use the likelihood of being true bugs to prioritize RV *violations*, which is orthogonal and complementary to prioritizing properties. The evolution-aware RV techniques proposed by Legunsen et al. [59], [60] can reduce developer manual inspection effort but only work across multiple program versions. RVPRIO does not require multiple program versions and is complementary to evolution-aware RV techniques. Lastly, coming up with better properties, either manually or automatically, remains an active research topic. We expect the benefits of RVPRIO to be applicable until perfect properties (those for which every violation is guaranteed to be a true bug) are available.

B. Machine Learning in Software Engineering

The research on using machine learning in software engineering has a rich history. Examples of software engineering problems to which machine learning was applied include (1) predicting fault-prone or costly-to-maintain software system components based on historical information [42], [79], [96], (2) classifying field executions as passing or failing runs [30], [116], (3) duplicate bug report detection [99], [100],

[108], (4) bug localization [48], [67], (5) code search, code completion, code mining, code clone detection and code synthesis [27], [55], [65], [89], [90], [110], [115], (6) learning how to apply patches [105], [106], (7) prioritizing test programs for compilers [16], and (8) classifying warnings from SCA tools [2], [34]–[36], [46], [49]–[52], [69], [73], [88], [92], [95], [104], [107], [114], [117]. We are the first to apply machine learning to prioritizing and classifying violations from RV of test executions, for reducing developer manual inspection effort. Among the prior work on machine learning in software engineering, RVPRIO is most related to the work on classifying warnings from static analysis tools, which we discuss next.

C. Classifying Static Analysis Warnings

There is a long line of work on using machine learning to classify warnings from Static Code Analysis (SCA) tools.

Koc et al. [50], [51] empirically evaluated the effectiveness of four families of machine-learning approaches for classifying SCA tool warnings from 14 programs, using hand-crafted features. They found that recurrent neural networks performed best. In contrast, we found ensemble techniques worked best for classifying RV violations, in line with future work that Koc et al. proposed [50]—they conjectured that ensemble techniques may achieve better accuracy.

Kim and Ernst [49] used the fix history of SCA tool warnings to prioritize new warnings; if more warnings of a particular kind were usually fixed quickly in the past, then new warnings of that kind are ranked higher. Also, in comparing algorithms that rank warnings, Allier et al. [2] found that the best ranking algorithms are based on the history of past warning and their location in the code. Our results with locations of violations in the code are already promising, and it would be interesting to see how violation history would improve the precision of RVPRIO.

Xypolytos et al. [114], Meng et al. [69] and Ribiero et al. [92] proposed approaches for combining output from multiple SCA tools when prioritizing SCA tool warnings. We only prioritize warnings from one RV tool, JavaMOP, and leave as future work to investigate the impact of combining output from multiple RV tools. In their work, Ribiero et al. [92] trained an ensemble classifier (they used AdaBoost) because they neither considered project history nor performed deep code analysis when selecting features. Interestingly, we also do not consider project history, and the classifiers that performed best were based on ensemble techniques (Section IV-C).

Wang et al. [107] studied whether there is a "golden set" of features that make classifiers effective in classifying SCA tool warnings. By investigating 116 features used in 10 papers that used machine learning to classify SCA tool warnings, they found a set of 23 golden features that were commonly used, and which were effective for warning classification. Some of the features that we used are in the final set of 23 golden features that they identified. As the research on classifying RV violations progresses, it will be interesting to see whether a corresponding set of golden features emerges.

Several techniques for classifying warnings of SCA tools are user guided, leveraging prior decisions to improve precision of reports [88], [104]. The labels of violations in our dataset reflect the decisions that real RV tool users made in prior work. An interesting future direction is to use RVPRIO together with evolution-aware RV techniques [59], [60], so that user decisions from prior versions are taken into account when prioritizing and classifying violations in future versions.

Jung et al. [46] proposed Airac, to classify as true bugs alarms whose true-bug probability was above a threshold, and prioritize the warnings presented to the user in order of their true-bug probability. We have implemented the same ideas for classification and prioritization in RVPRIO, but for classifying violations from RV. We also performed a case study with RVPRIO, which was not done for Airac, as far as we know.

Concerning using only a subset of possible features when building classifiers for warnings from SCA tools, Ruthruff et al. [95] used a screening methodology to remove features with low predictive power. They found that classifiers built with the reduced set of features were comparable to classifiers built with the full set of features in terms of accuracy, and were less expensive to use on large systems. Their result gives us more confidence in the performance of the classifiers we obtained after feature selection (Section III-A).

VIII. Conclusions

Runtime Verification (RV) during testing was recently shown to be effective for detecting many more bugs than testing alone. But, using RV is laborious and time-consuming—RV tools often generate many violations that developers have to manually inspect.

We presented RVPRIO, the first automated approach to reduce developer manual inspection effort by ranking violations in order of likelihood of being true bugs. RVPRIO uses machine-learning classifiers to prioritize violations. We trained RVPRIO on the largest publicly-available dataset containing 1,170 previously labeled RV violations. Results on the labeled dataset show that RVPRIO ranked 88.10% of true bugs in the top 25% of the prioritized list. We also conducted a case study to apply RVPRIO to unlabeled violations and obtained similarly good results. From inspecting these new violations, we found 29 previously unknown bugs in 7 projects and two bugs in the formally specified properties that we used. RVPRIO opens a new research direction on reducing developer manual inspection effort for performing RV during testing, and we highlighted some possible future work in this direction (see Section VI). Our data is publicly available online at https://github.com/brenomiranda/rvprio.

Acknowledgments. The work in this paper was partially funded by INES 2.0, FACEPE grants PRONEX APQ 0388-1.03/14 and APQ-0399-1.03/17, CAPES grant 88887.136410/2017-00, CNPq grant 465614/2014-0, as well as US National Science Foundation grants CNS-1646305 and CNS-1740916.

REFERENCES

[1] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Adding trace matching with free variables to AspectJ," in *OOPSLA*, 2005.

[2] S. Allier, N. Anquetil, A. Hora, and S. Ducasse, "A framework to compare alert ranking algorithms," in *Reverse Engineering*, 2012.

[3] "Apache projects list," 2019, https://builds.apache.org/.

[4] "Apache Commons Lang," 2019, https://commons.apache.org/proper/commons-lang/.

[5] M. Arnold, M. Vechev, and E. Yahav, "QVM: An efficient runtime for detecting defects in deployed systems," in *OOPSLA*, 2008.

[6] H. Barringer, D. Rydeheard, and K. Havelund, "Rule systems for run-time monitoring: from Eagle to RuleR," *JLC*, vol. 20, no. 3, 2010.

[7] N. E. Beckman and A. V. Nori, "Probabilistic, modular and scalable inference of typestate specifications," in *PLDI*, 2011.

[8] E. Bodden, "MOPBox: A library approach to runtime verification," in *RV Tool Demo*, 2011.

[9] E. Bodden, L. Hendren, P. Lam, O. Lhoták, and N. A. Naeem, "Collaborative runtime verification with Tracematches," in *RV*, 2007.

[10] E. Bodden, L. J. Hendren, and O. Lhoták, "A staged static program analysis to improve the performance of runtime monitoring," in *ECOOP*, 2007.

[11] E. Bodden, P. Lam, and L. Hendren, "Finding programming errors earlier by evaluating runtime monitors ahead-of-time," in *FSE*, 2008.

[12] "ByteArrayOutputStream_FlushBeforeRetrieve property," 2016, https://bit.ly/2KU5g5A.

[13] G. Chandrashekar and F. Sahin, "A survey on feature selection methods," *Computers & Electrical Engineering*, vol. 40, no. 1, 2014.

[14] D. Chen, Y. Zhang, R. Wang, X. Li, L. Peng, and W. Wei, "Mining universal specification based on probabilistic model," in *SEKE*, 2015.

[15] F. Chen and G. Roşu, "MOP: An efficient and generic runtime verification framework," in *OOPSLA*, 2007.

[16] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing," in *ICSE*, 2017.

[17] "Collections_SynchronizedCollection property," 2019, https://bit.ly/2MiUtCZ.

[18] "Synchronize before looping over synchronized collection," 2020, https://github.com/apache/commons-lang/pull/399.

[19] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller, "Generating test cases for specification mining," in *ISSTA*, 2010.

[20] M. Dash and H. Liu, "Feature selection for classification," *Intelligent Data Analysis*, vol. 1, no. 1-4, 1997.

[21] N. Decker, J. Harder, T. Scheffel, and D. Schmitz, Malteand Thoma, "Runtime monitoring with union-find structures," in *TACAS*, 2016.

[22] U. Erlingsson and F. B. Schneider, "IRM enforcement of Java stack inspection," in *IEEE S&P*, 2000.

[23] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim, "Do we need hundreds of classifiers to solve real world classification problems?" *Journal of Machine Learning Research*, vol. 15, no. 1, 2014.

[24] V. Forejt, M. Kwiatkowska, D. Parker, H. Qu, and M. Ujma, "Incremental runtime verification of probabilistic systems," in *RV*, 2012.

[25] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, 2001.

[26] M. Gabel and Z. Su, "Online inference and enforcement of temporal properties," in *ICSE*, 2010.

[27] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *FSE*, 2016.

[28] L. Guo, Y. Ma, B. Cukic, and Harshinder Singh, "Robust prediction of fault-proneness by random forests," in *ISSRE*, 2004.

[29] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik, "Gene selection for cancer classification using support vector machines," *Machine Learning*, vol. 46, no. 1-3, 2002.

[30] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil, "Applying classification techniques to remotely-collected program execution data," in *ESEC/FSE*, 2005.

[31] K. Havelund, D. Peled, and D. Ulus, "First-order temporal logic monitoring with BDDs," in *FMCAD*, 2017.

[32] K. Havelund and G. Roşu, "Monitoring Java programs with Java PathExplorer," in *RV*, 2001.

[33] ——, "Monitoring programs using rewriting," in *ASE*, 2001.

[34] S. Heckman, "A systematic model building process for predicting actionable static analysis alerts," Ph.D. dissertation, NCSU, 2009.

[35] S. Heckman and L. Williams, "On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques," in *ESEM*, 2008.

[36] ——, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *IST*, vol. 53, no. 4, 2011.

[37] S. Hussein, P. Meredith, and G. Roşu, "Security-policy monitoring and enforcement with JavaMOP," in *PLAS*, 2012.

[38] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: With Applications in R*, 2014.

[39] C. Jaspan, M. Jorde, A. Knight, C. Sadowski, E. K. Smith, C. Winter, and E. Murphy-Hill, "Advantages and disadvantages of a monolithic repository: A case study at Google," in *ICSE SEIP*, 2018.

[40] "JavaMOP4," 2015, http://fsl.cs.illinois.edu/index.php/JavaMOP4.

[41] "JavaMOP4 Syntax," 2015, http://fsl.cs.illinois.edu/index.php/JavaMOP4_Syntax.

[42] Jianhui Tian, A. Porter, and M. V. Zelkowitz, "An improved classification tree analysis of high cost modules based upon an axiomatic definition of complexity," in *ISSRE*, 1992.

[43] D. Jin, P. O. Meredith, D. Griffith, and G. Roşu, "Garbage collection for monitoring parametric properties," in *PLDI*, 2011.

[44] D. Jin, P. O. Meredith, C. Lee, and G. Roşu, "JavaMOP: Efficient parametric runtime monitoring framework," in *ICSE Demo*, 2012.

[45] D. Jin, P. O. Meredith, and G. Roşu, "Scalable parametric runtime monitoring," Computer Science Dept., UIUC, Tech. Rep., 2012.

[46] Y. Jung, J. Kim, J. Shin, and K. Yi, "Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis," in *SAS*, 2005.

[47] M. Karaorman and J. Freeman, "jMonitor: Java runtime event specification and monitoring library," in *RV*, 2004.

[48] D. Kim, Y. Tao, S. Kim, and A. Zeller, "Where should we fix this bug? a two-phase recommendation model," *TSE*, vol. 39, no. 11, 2013.

[49] S. Kim and M. D. Ernst, "Which warnings should I fix first?" in *ESEC/FSE*, 2007.

[50] U. Koc, S. Wei, J. S. Foster, M. Carpuat, and A. A. Porter, "An empirical assessment of machine learning approaches for triaging reports of a Java static analysis tool," in *ICST*, 2019.

[51] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter, "Learning a classifier for false positive error reports emitted by static code analysis tools," in *MAPL*, 2017.

[52] T. Kremenek and D. Engler, "Z-ranking: Using statistical analysis to counter the impact of static analysis approximations," in *SAS*, 2003.

[53] R. Krishna, T. Menzies, and L. Layman, "Less is more: Minimizing code reorganization using XTREE," *IST*, vol. 88, 2017.

[54] I. Krka, Y. Brun, and N. Medvidovic, "Automatic mining of specifications from invocation traces and method invariants," in *FSE*, 2014.

[55] N. Kushman and R. Barzilay, "Using semantic unification to generate regular expressions from natural language," in *ACL*, 2013.

[56] C. Le Goues and W. Weimer, "Specification mining with few false positives," in *TACAS*, 2009.

[57] C. Lee, F. Chen, and G. Roşu, "Mining parametric specifications," in *ICSE*, 2011.

[58] C. Lee, D. Jin, P. O. Meredith, and G. Roşu, "Towards categorizing and formalizing the JDK API," Computer Science Dept., UIUC, Tech. Rep., 2012.

[59] O. Legunsen, D. Marinov, and G. Roşu, "Evolution-aware monitoring-oriented programming," in *ICSE NIER*, 2015.

[60] O. Legunsen, Y. Zhang, M. Hadzi-Tanovic, G. Rosu, and D. Marinov, "Techniques for evolution-aware runtime verification," in *ICST*, 2019.

[61] O. Legunsen, "Evolution-Aware Runtime Verification," Ph.D. dissertation, University of Illinois at Urbana-Champaign, USA, 2019.

[62] O. Legunsen, N. A. Awar, X. Xu, W. U. Hassan, G. Roşu, and D. Marinov, "How effective are existing Java API specifications for finding bugs during runtime verification?" *ASEJ*, vol. 26, no. 4, 2019.

[63] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov, "How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications," in *ASE*, 2016.

[64] C. Lemieux, D. Park, and I. Beschastnikh, "General LTL specification mining," in *ASE*, 2015.

[65] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kočiský, F. Wang, and A. Senior, "Latent predictor networks for code generation," in *ACL*, 2016.

[66] V. Lopez, A. Fernandez, and F. Herrera, "On the importance of the validation technique for classification with imbalanced datasets: Addressing covariate shift when data is skewed," *Information Sciences*, vol. 257, 2014.

[67] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using Latent Dirichlet Allocation," *IST*, vol. 52, no. 9, 2010.

[68] Q. Luo, Y. Zhang, C. Lee, D. Jin, P. O. Meredith, T. F. Serbanuta, and G. Roşu, "RV-Monitor: Efficient parametric runtime verification with simultaneous properties," in *RV*, 2014.

[69] N. Meng, Q. Wang, Q. Wu, and H. Mei, "An approach to merge results of multiple static analysis tools," in *ICQS*, 2008.

[70] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *ASEJ*, vol. 17, no. 4, 2010.

[71] P. Meredith and G. Roşu, "Efficient parametric runtime verification with deterministic string rewriting," in *ASE*, 2013.

[72] P. Meredith, D. Jin, F. Chen, and G. Roşu, "Efficient monitoring of parametric context-free patterns," in *ASE*, 2008.

[73] T. Muske and A. Serebrenik, "Survey of approaches for handling static analysis alarms," in *SCAM*, 2016.

[74] S. Navabpour, C. W. W. Wu, B. Bonakdarpour, and S. Fischmeister, "Efficient techniques for near-optimal instrumentation in time-triggered runtime verification," in *RV*, 2011.

[75] A. C. Nguyen and S.-C. Khoo, "Extracting significant specifications from mining through mutation testing," in *ICFEM*, 2011.

[76] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan, "Mining preconditions of APIs in large-scale code corpus," in *FSE*, 2014.

[77] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, 2011.

[78] "PMD: An extensible cross-language static code analyzer," 2019, https://pmd.github.io/.

[79] A. A. Porter and R. W. Selby, "Empirically guided software development using metric-based classification trees," *IEEE Software*, vol. 7, no. 2, 1990.

[80] R. Potvin and J. Levenberg, "Why Google stores billions of lines of code in a single repository," *CACM*, vol. 59, no. 7, 2016.

[81] M. Pradel, P. Bichsel, and T. R. Gross, "A framework for the evaluation of specification miners based on finite state machines," in *ICSM*, 2010.

[82] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," in *ASE*, 2009.

[83] ——, "Leveraging test generation and specification mining for automated bug detection without false positives," in *ICSE*, 2012.

[84] "FSL Specification Database," 2016, https://runtimeverification.com/monitor/propertydb.

[85] R. Purandare, M. B. Dwyer, and S. Elbaum, "Monitor optimization via stutter-equivalent loop transformation," in *OOPSLA*, 2010.

[86] ——, "Optimizing monitoring of finite state properties through monitor compaction," in *ISSTA*, 2013.

[87] D. Radjenovic, M. Hericko, R. Torkar, and A. Zivkovic, "Software fault prediction metrics: A systematic literature review," *IST*, vol. 55, no. 8, 2013.

[88] M. Raghothaman, S. Kulkarni, K. Heo, and M. Naik, "User-guided program reasoning using Bayesian inference," in *PLDI*, 2018.

[89] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," in *POPL*, 2015.

[90] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *PLDI*, 2014.

[91] G. Reger, H. Barringer, and D. Rydeheard, "A pattern-based approach to parametric specification mining," in *ASE*, 2013.

[92] A. Ribeiro, P. Meirelles, N. Lago, and F. Kon, "Ranking source code static analysis warnings for continuous monitoring of FLOSS repositories," in *OSS*, 2018.

[93] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *TSE*, vol. 39, no. 5, 2013.

[94] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *ICSM*, 1999.

[95] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: An experimental approach," in *ICSE*, 2008.

[96] R. W. Selby and A. A. Porter, "Learning from examples: generation and evaluation of decision trees for software resource analysis," *TSE*, vol. 14, no. 12, 1988.

[97] "SortedSet_Comparable property," 2019, https://bit.ly/2N8ck1v.

[98] "addAll() bug in SortedSet_Comparable.mop & TreeSet_Comparable.mop," 2020, https://github.com/runtimeverification/property-db/issues/7.

[99] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *ICSE*, 2010.

[100] A. Sureka and P. Jalote, "Detecting duplicate bug report using character n-gram-based features," in *APSEC*, 2010.

[101] A. Tharwat, "Classification assessment methods," *Applied Computing and Informatics*, 2018.

[102] S. Thummalapenta and T. Xie, "Alattin: Mining alternative patterns for detecting neglected conditions," in *ASE*, 2009.

[103] "TreeSet_Comparable property," 2019, https://bit.ly/2P1QEXt.

[104] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin, "ALETHEIA: Improving the usability of static security analysis," in *CCS*, 2014.

[105] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *ICSE*, 2019.

[106] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *ASE*, 2018.

[107] J. Wang, S. Wang, and Q. Wang, "Is there a "golden" feature set for static warning identification?: An experimental evaluation," in *ESEM*, 2018.

[108] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *ICSE*, 2008.

[109] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," in *ASE*, 2009.

[110] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *ASE*, 2016.

[111] M. Woniak, M. Graña, and E. Corchado, "A survey of multiple classifier systems as hybrid systems," *Information Fusion*, vol. 16, 2014.

[112] C. W. W. Wu, D. Kumar, B. Bonakdarpour, and S. Fischmeister, "Reducing monitoring overhead by integrating event- and time-triggered techniques," in *RV*, 2013.

[113] Q. Wu, G. Liang, Q. Wang, T. Xie, and H. Mei, "Iterative mining of resource-releasing specifications," in *ASE*, 2011.

[114] A. Xypolytos, H. Xu, B. Vieira, and A. M. T. Ali-Eldin, "A framework for combining and ranking static analysis tool findings based on tool performance statistics," in *QRS-C*, 2017.

[115] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *ICSE*, 2016.

[116] C. Yilmaz and A. Porter, "Combining hardware and software instrumentation to classify program executions," in *FSE*, 2010.

[117] U. Yuksel and H. Sozer, "Automated classification of static code analysis alerts: A case study," in *ICSM*, 2013.

[118] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language API documentation," in *ASE*, 2009.