An Empirical Assessment of Machine Learning Approaches for Triaging Reports of a Java Static Analysis Tool

Ugur Koc[*] Shiyi Wei[†] Jeffrey S. Foster[‡] Marine Carpuat[*] Adam A. Porter[*]
[*]Department of Computer Science, University of Maryland, College Park, USA
{ukoc, marine, aporter}@cs.umd.edu
[†]Department of Computer Science, University of Texas, Dallas, USA
swei@utdallas.edu
[‡]Department of Computer Science, Tufts University, Medford, MA, USA
jfoster@cs.tufts.edu

***Abstract*—Despite their ability to detect critical bugs in software, developers consider high false positive rates to be a key barrier to using static analysis tools in practice. To improve the usability of these tools, researchers have recently begun to apply machine learning techniques to classify and filter false positive analysis reports. Although initial results have been promising, the long-term potential and best practices for this line of research are unclear due to the lack of detailed, large-scale empirical evaluation. To partially address this knowledge gap, we present a comparative empirical study of four machine learning techniques, namely hand-engineered features, bag of words, recurrent neural networks, and graph neural networks, for classifying false positives, using multiple ground-truth program sets. We also introduce and evaluate new data preparation routines for recurrent neural networks and node representations for graph neural networks, and show that these routines can have a substantial positive impact on classification accuracy. Overall, our results suggest that recurrent neural networks (which learn over a program's source code) outperform the other subject techniques, although interesting tradeoffs are present among all techniques. Our observations provide insight into the future research needed to speed the adoption of machine learning approaches in practice.**

***Index Terms*—static analysis, false positive classification, machine learning**

I. Introduction

Static analysis (SA) tools are designed to detect errors that might jeopardize the security and performance of software applications. Unfortunately, SA tools frequently generate large numbers of false positives results, i.e., non-errors incorrectly labeled as errors. Because developers must manually resolve each SA report, many regard this issue as a key barrier to using SA tools in practice [32]. That is, developers perceive the cost of painstakingly analyzing hundreds or even thousands of error reports that ultimately turn out be false to outweigh the cost of missing some potential errors.

To address this problem, SA researchers have, in fact, proposed numerous tweaks and improvements (e.g., [10], [20], [34], [53]). While these efforts have had a positive effect, in practice, false positives are still too common [40]. One reason for this is that the symptoms of false positives are numerous and varied and, therefore, hard to detect with simple predefined checkers. Recently, researchers have applied machine learning (ML) approaches to triage SA reports by ranking or filtering so that developers can focus on the reports that are likely to be true positives. Their hope is that ML approaches could learn patterns that lead to false positives, and that are hard to detect with traditional static analysis approaches, thereby greatly improving the practical cost and benefit ratio of using SA tools [33], [35], [36], [50], [70].

一些关于使用ML辅助静态分析的研究

To date, most research in this area has used ML techniques with hand-engineered features to classify and filter false positives [22], [23], [61], [69]. These approaches focus on learning observable features of the static analysis results. They also tend to include black-box observations of the target programs. For example, bug type or rule violation type is one commonly used feature in several recent approaches [23], [61], [69]. Potential limitations of these approaches include that feature identification often relies on manual and time-consuming investigations by experts, and that, by design, the approach ignores the deep structure of the source code being analyzed, inevitably leading to a loss of accuracy. To address these limitations, Koc *et al.* [35] experimented with neural network-based learning approaches, which could potentially capture source code-level characteristics that may have led to false positives. Their evaluation on synthetic benchmarks showed that a specific type of recurrent neural network significantly improved classification accuracy, compared to a Bayesian inference-based approach. Given the limited data set involved in that study, however, further study is called for.

Overall, while existing research suggests the benefits of applying ML algorithms to classify SA results, there are important open research questions to be addressed before machine learning algorithms are likely to be routinely applied to this use case. First and foremost, there has been relatively little extensive empirical evaluation of different ML algorithms. Such empirical evaluation is of great practical importance for understanding the tradeoffs and requirements of ML algorithms. Second, the effectiveness and generalizability of the features used and data preparation techniques needed for different ML techniques have not been well-investigated for actual usage scenarios. Third, there is also need for larger, real-world program datasets to better validate the findings of prior work which was largely conducted on synthetic benchmarks. These open problems leave uncertainty as to which approaches to use and when to use them in practice.

To partially address these limitations, in this work, we describe a systematic, comparative study of multiple ML approaches for classifying SA results[1]. Our study makes several key contributions. First, we create a real-world, ground-truth benchmark for the experiments, consisting of 14 Java programs covering a wide range of application domains and 400 vulnerability reports from a widely used SA tool for Java (Section III). Second, we introduce key data preparation routines for applying neural networks to this problem (Section II). Third, we compare the effectiveness of four families of ML approaches: hand-engineered features, bag of words, recurrent neural networks, and graph neural networks, with different combinations of data preparation routines (Section IV).

1. 我们创建了测试标准，从JavaProgram中和广泛使用的Java工具中提取漏洞报告
2. 介绍了将神经网络应用此问题的关键数据准备程序
3. 比较四纵ML方法的有效性

Our experimental results provide significant insights into the performance and applicability of the ML algorithms and data preparation techniques in two different real-world application scenarios we studied. First, we observe that the recurrent neural networks perform better compared to the other approaches. Second, with more precise data preparation, we achieved large performance improvements over the state-of-the-art [35]. Furthermore, with the two application scenarios we studied, we demonstrated that the data preparation for neural networks has a significant impact on the performance and generalizability of the approaches, and different data preparation techniques should be applied in different application scenarios (Section V).

RNN表现最好，数据准备对于结果有很大影响

II. Adapting ML to Classify False Positives

In this section, we discuss the four ML approaches we studied: learning with hand-engineered features (HEF), bag of words (BoW), recurrent neural networks (RNN), and graph neural networks (GNN). We first provide brief background information with some examples of their use in software engineering. Then we describe how we apply them to the SA false positive classification task with data preparation routines we developed or adopted. HEF can be regarded as the state-of-the-art ML application for this problem [64]. However, by design, they cannot include the deep structure of the source code being analyzed. The other three approaches add an increasing amount of structural information as we move from BoW to GNN. To the best of our knowledge, BoW and GNN have not been used to solve this problem before, while RNN has been used in a recent case study [35]. Furthermore, previous work has not focused on the details of the data preparation routines or their effects on the training performance and generalizability.

Detecting false positives can be framed as a standard binary classification problem [55]. Given an input $\vec{x}$, e.g., a point in a high dimensional space $\mathbb{R}^D$, the classifier produces an output $y = f_\theta(\vec{x})$, where $y = 1$ for false positives, $y = 0$ otherwise. Constructing such a classifier requires defining input vector $\vec{x}$ that captures features of programs that might help detect false positives. We also need to select a function f_θ, as different families of functions encode different inductive biases and assumptions about how to predict outputs for new inputs. Once these two decisions have been made, the classifier can be trained, by estimating its parameters θ from a large set of known false positives and true positives $\{(x_1, y_1) \dots (x_N, y_N)\}$.

A. Learning with Hand-engineered Features

Background: A feature vector $\vec{x}$ can be constructed by asking experts to list measurable properties of the program and SA report that might be indicative of true positives or false positives. Each property can then be represented numerically by one or more elements in $\vec{x}$. Hand-engineered features have been defined to classify SA false positives in existing work [22], [23], [61], [69]. Tripp *et al.* [61] identified features to filter false cross-site scripting (XSS) vulnerability reports for JavaScript programs. These features are: (1) *source identifier* (e.g., `document.location`), (2) *sink identifier* (e.g., `window.open`), (3) *source line number*, (4) *sink line number*, (5) *source URL*, (6) *sink URL*, (7) *external objects* (e.g., flash), (8) *total results*, (9) *number of steps* (flow milestones comprising the witness path), (10) *analysis time*, (11) *number of path conditions*, (12) *number of functions*, (13) *rule name*, (14) *severity*. The first seven features are lexical, the next five are quantitative, and last two are security-specific. Note that, identifying these features requires expertise in web application security and JavaScript.

Our adaptation: In this work, we adapted the original feature set from Tripp *et al.* [61] by dropping features *source URL, sink URL* and *external objects* as they do not appear in Java applications (our datasets consist of Java programs, Section III), and extending it with two features extracted from SA reports that might improve the detection of false positives: *confidence* of the analyzer, which is designed to measure the likelihood of a report being a true positive, and *number of classes* referred in the error trace. We conjecture that longer error traces with references to many classes might indicate imprecision in the analysis, thus suggesting a higher chance of false positives.

Once the feature representations are defined, a wealth of classifiers f_θ and training algorithms can be used to learn how to make predictions. Since feature vectors $\vec{x}$ encode rich knowledge about the task, classifiers f_θ that compute simple combinations of these features can be sufficient to train good models quickly. However, defining diverse features that capture all variations that might occur in different datasets is challenging and requires human expertise.

Next, we explore how to represent program source code for more complex ML approaches that can implicitly learn feature representations.

[1] The experimental infrastructure of this study (including all raw data) is available at: https://bitbucket.org/ugur_koc/mangrove

```
1 EXPR 164 {
2 O reference;
3 V "v3 = com.mangrove.utils.DBHelper.conn";
4 T "Ljava/sql/Connection";
5 S "com/mangrove/utils/DBHelper.java":15,0;
6 DD 166;
7 CF 166;
8 ...}
```

Fig. 1. Sample PDG Node (simplified for presentation)

B. Program Slicing for Summarization

Background: Real-world programs are large (see Table I), and learning directly from such data is challenging since many sections of the code are unlikely to pertain to false positives and are likely to introduce noise for certain ML approaches. To address this difficulty, Koc *et al.* [35] computed the backward slice of the programs being analyzed starting from the reported source line [65]. The rationale for this choice is that the backward slice is a good summary of programs for the SA report classification task as it includes all the statements that may affect the behavior at the reported line.

Our adaptation: In this work, we also use program slicing as a pre-summarization step for BoW, RNN, and GNN approaches. Specifically, we used Joana [29], a program analysis framework for Java, for computing backward slices. The first step is determining the entry points from which the program starts to run. For our problem, we first find the call hierarchy of the method containing the error line, then in this hierarchy identify the methods that can be invoked by user to set as the entry points. Such methods can be APIs if the program is a library, or the main method (which is the default entry point), or test cases. Next, we compute the program dependency graph (PDG) which consists of PDG nodes denoting the program locations that are reachable from the entry points. Then, we identify the PDG node(s) that appear in the reported source line. Finally, we compute the backward slice from that line to the entry point(s).

Java切词工具

Figure 1 shows an example PDG node. Line 1 shows the kind and ID of the node, which are `EXPR` and 164, respectively. At line 2, we see the operation is a `reference`. At line 3, `V` denotes the value of the bytecode statement in *WALA* IR[2]. At line 4, `T` is the type of the statement (here, the `Connection` class in `java.sql`). Lastly, there is a list of outgoing dependency edges. `DD` and `CF` at lines 7 and 8 denote that this node has a data dependency edge and a control-flow edge, respectively, to the node with ID 166. In the following discussion, we will refer to these fields of PDG node.

C. Bag of Words

比较单词集合（不计顺序）的相似性

Background: How can we represent a program slice as a feature vector that contains useful information to detect false positives? We take inspiration from text classification problems, where classifier inputs are natural language documents and "Bag of Words" (BoW) features provide simple yet effective representations [16]. BoW represents a document as a multiset of the words found in the document, ignoring their order. The resulting feature vector $\vec{x}$ for a document has as many entries as words in the dictionary, and each entry indicates whether a specific word exists in the document.

BoW has been used in software engineering as an information retrieval technique to solve problems such as duplicate report detection [59], bug localization [41], and code search [68]. Such applications often use natural language descriptions provided by humans (developers or users). To our knowledge, BoW has not been used to classify SA reports.

我们用了两种不同的BoW，一种是单词的位置，1/0表示是/否出现在程序中

第二种是单词出现的频率

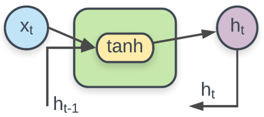
Our adaptation: In our experiments, we used two variations of BoW. The first variation checks the occurrence of words, which leads to a binary feature vector representation, where the features are the words. 1 means that the corresponding word is in a program, and 0 means it is not. The second variation counts the frequency of words, which leads to an integer feature vector, where each integer indicates how many times the corresponding word occurs. In our setting, "words" correspond to tokens extracted from program slices using data preparation routines introduced in Section II-D.

Similar to the HEF approach, once the feature vector representations are created, any classification algorithm can be used for training. For a fixed classifier, training with BoW often takes longer than learning with HEF because the feature space (i.e., the dictionary) is usually significantly larger.

D. Recurrent Neural Networks

Background: BoW features ignore order. For text classification, recurrent neural networks [14], [24], [42] have emerged as a powerful alternative approach that views text as an (arbitrary-length) sequence of words and automatically learn vector representations for each word in the sequence [16].

RNNs process a sequence of words with arbitrary-length $X = \langle x_0, x_1, \ldots, x_t, \ldots, x_n \rangle$ from left to right, one position at a time. For each position t, RNNs compute a feature vector h_t as a function of the observed input x_t and the representation learned for the previous position h_{t-1}, i.e, $h_t = \text{RNN}(x_t, h_{t-1})$. Once the sequence has been read, the average vectors $\langle h_0, h_1, \ldots, h_n \rangle$ is used as input to a logistic regression classifier.



RNNs can take different forms. A standard RNN unit is illustrated in the figure on the left. During training, the parameters of the logistic regression classifier and of the *RNN* function are estimated jointly. As a result, the vectors h_t can be viewed as feature representations for x_t that are learned from data, implicitly capturing relevant context knowledge about the sequence prefix $\langle x_0, \ldots x_{t-1} \rangle$ due to the structure of the RNN. Unlike HEF or BoW, the feature vectors h_t are directly optimized for the classification task. This advantage comes at the cost of interpretability since the values of h_t are much harder for humans to interpret than the BoW or HEF.

Recently, researchers have begun to use RNNs to solve SE task such as code completion [8], and code synthesis [37],

[2] Joana uses the intermediate representation from the T.J. Watson Libraries for Analysis (*WALA*) [63].

[39]. In a recent paper, Koc *et al.* [35] conducted a case study using Long Short-term Memories (LSTM) [14], [24], a popular kind of RNN, for classifying SA false positives.

Our adaptation: In this work, we study LSTMs as well, as they are well suited to modeling long sequences [56]. To use LSTM, we need to transform program slices into sequences of tokens, which we achieve with four sets of transformations. We denote each transformation as T_x for some x so we can refer to it later in the paper. We list the transformations in order of complexity, and a transformation is applied only after applying all of the other, less complex transformations.

数据清理：删除一些节点（不属于程序）

1) Data Cleansing and Tokenization (T_{cln}): This set of transformations remove certain PDG nodes and perform basic tokenization. First, they remove nodes of certain kinds (i.e., `formal_in`, `formal_out`, `actual_in`, `actual_out`), or whose value fields contain any of the phrases: `many2many`, `UNIQ`, `⟨init⟩`, `immutable`, `fake`, `_exception_`, or whose class loader is *Primordial*, which means this class is not part of programs source code. These nodes are removed because they do not provide useful information for the learning. Some of them do not even inhibit anything from the actual content of programs, but rather they are in the PDG to satisfy static single assignment form[3]. For instance, the nodes with type `NORM` and operation `compound` do not have bytecode instructions from the program in their value field (only the phrase `many2many`). Second, they extract tokens from paths of classes and methods by splitting them by '.' or '/'.

抽象文字和字符串，例如两位数字用N2，三位数字用N3，更多位数字用N+,接着提取字符串列表并进行编号化，如STR1，STR2等

2) Abstracting Numbers and String Literals (T_{ans}): These transformations replace numbers and string literals that appear in a program slice with abstract values. We hypothesize that these transformations will make learning more effective by reducing the vocabulary of a given dataset and will help us to train more generalizable models.

First, two digit numbers are replaced with *N2*, three digit numbers are with *N3*, and numbers with four or more digit are with *N4+*. We apply similar transformations for negative numbers and numbers in scientific notation. Next, we extract the list of string literals and replace each of them with the token `STR` followed by a unique number. For example, the first string literal in the list will be replaced with `STR 1`.

许多程序员使用一组通用的短变量作为标识符如i,j,counter等，它对我们分析有意义，而其他变量名对程序分析无意义

3) Abstracting Program-specific Words (T_{aps}): Many programmers use a common, small set of words as identifiers, e.g., `i`, `j`, and `counter` are often used as integer variable identifiers. We expect that such commonplace identifiers are also helpful for our learning task. On the other hand, programmers might use identifiers that are program- or domain-specific, and hence do not commonly appear in other programs. Learning these identifiers may not be useful for classifying SA reports in other programs.

因此通过短语UNK标记一些无意义单词

Therefore, T$_{aps}$ abstracts away certain words from the dataset that occur less than a certain amount of time, or that only occur in a single program, by replacing them with phrase *UNK*. Similar to T$_{ans}$, these transformations are expected to improve the effectiveness by reducing the vocabulary size and generalizability via abstractions.

将camelCase和snake_case标识符拆分为组成单词

4) Extracting English Words From Identifiers (T_{ext}): Many identifiers are composed of multiple English words. For example, the `getFilePath` method from the Java standard library consists of three English words: `get`, `File`, and `Path`. To make our models more generalizable and to reduce the vocabulary size, we split any `camelCase` or `snake_case` identifiers into their constituent words.

这些转变的作用之前未被调查

To the best of our knowledge, the effects of these transformations have not been thoroughly studied in the past, although transformations similar to T$_{cln}$, T$_{ans}$, and T$_{aps}$ were used by Koc *et al.* [35]. We further improved and extended the transformations for mapping string literals and numbers with generic placeholders (e.g., `STR 1`, `N1`), splitting paths of classes and methods, and removing certain PDG nodes to improve the effectiveness and generalizability.

E. Graph Neural Networks

RNN不能表示程序的图结构，因此引入GNN

Background: With RNN, we represent programs as a sequence of tokens. However, programs have a more complex structure that might be better represented with a graph. To leverage such structure, we explore graph neural networks which compute vector representations for nodes in a graph using information from neighboring nodes [18], [57]. The graphs are of the form $G = \langle N, E \rangle$, where $N = n_0, n_1, \ldots, n_i$ is the set of nodes, and $E = e_1, e_2, \ldots, e_j$ is the set of edges. Each node i is represented with a vector h_i, which captures learned features of the node in the context of the graph.

The edges are of the form $e = \langle type, source, dest \rangle$, where *type* is the type of the edge and *source* and *dest* are the IDs of the source and destination nodes, respectively. The vectors h_i are computed iteratively, starting with arbitrary values at time $t = 0$, and incorporating information from neighboring nodes $\text{NBR}(n_i)$ at each time step t, i.e, $h_i^{(t)} = f(n_i, h_{\text{NBR}(n_i)}^{(t-1)})$. The function f is defined as a neural network.

我们使用了GNN的变体GGNN，引入三种初始化输入节点表示：

1）种类、操作和类型字段（参照图一）

Our adaptation: In our study, we focus on a variation of GNNs called Gated Graph Neural Networks (GGNN) [38]. GGNNs have gated recurrent units and enable initialization of the node representation. GGNNs have been used to learn properties about programs [3], [38], but have not been applied to classify SA false positives or to learn from program slices. To adapt GGNNs to our problem, we introduce three approaches for initializing the input node representations n_i.

1) Using Kind, Operation, and Type Fields (KOT): As the first representation, we only use the `Kind`, `Operation`, and `Type` fields of the PDG nodes. For the example node in Figure 1, the KOT node representation is V_{rep} =[`EXPR`, `reference`, `Ljava/sql/Connection`]

2) Extracting a Single Item in Addition to KOT (KOTI): In the second representation, in addition to KOT, we include one more item that usually comes from the `Value` field of the PDG node depending on the `Operation` field. For example, if the operation is `call`, or `entry`, or `exit`, we extract the identifier of the method that appears in the statement. The KOTI representation for the PDG node in Figure 1 is

[3] A property of the representation which requires that each variable is assigned exactly once, and every variable is defined before it is used [54].

V_{rep} =[EXPR, reference, Ljava/sql/Connection, object] (object is the extracted item, meaning that the reference is for an object).

3) Node Encoding Using Embeddings (Enc): In the third representation, we use word embeddings to compute a vector representation that accounts for the entire bytecode in the Value field (which has arbitrary number of words). To achieve this, we first perform pre-training using a bigger, unlabeled dataset to learn embedding vectors that capture more generic aspects of the words in the dictionary using the *word2vec* model [17], [44]. Then we take the average of the embedding vectors of the words that appear in the Value field as its representation, E_V. Finally, we create a node vector by concatenating E_V with the embedding vectors of Kind, Operation, and type, i.e., $V_{rep} = E_K + E_O + E_T + E_V$ where $+$ is the concatenation operation.

使用Word2Vec进行节点编码

III. Tool and Benchmarks

The SA tool we study is FindSecBugs [11] (version 1.4.6), a popular security checker for Java web applications. We configured FindSecBugs to detect cross-site scripting (XSS), path traversal (XPATH), and SQL, command, CRLF, and LDAP injections. We selected this subset of vulnerabilities because they share similarities in performing suspicious operations on safety-critical resources. Such operations are detected by the taint analysis implemented in FindSecBugs.

We used two benchmarks in our evaluation. The first is the OWASP Benchmark [48], which has been used to evaluate various SA tools in the literature [6], [35], [67]. In particular, we used the same programs as Koc *et al.* [35] so we could compare results. This benchmark contains 2371 SQL injection vulnerability reports, 1193 of which are labeled as false positives; the remaining reports are labeled as true positives.

我们使用了OWASP Benchmark和14个真实的程序

We constructed the second benchmark, consisting of 14 real-world programs. We ran FindSecBugs on these programs, and then manually labeled the resulting vulnerability reports as true or false positives. We chose these programs using the following criteria:

- We selected programs for which FindSecBugs generates *vulnerability reports*. To have the kinds of vulnerabilities we study, we observe that programs should perform database and LDAP queries, use network connections, read/write files, and/or execute commands.
- We chose programs that are *open source* because we need access to source code to apply our ML algorithms.
- We chose programs that are under *active development* and are *highly used*.
- Finally, we chose programs that are *small to medium size*, ranging from 5K to 1M lines of code (LoC). Restricting code size was necessary to successfully create the PDG which is used for program slicing [29].

Table I shows the details of the collected programs. Several programs have been used in past research: H2-DB and Hsqldb are from the Dacapo Benchmark [5] (the other Dacapo programs did not satisfy our selection criteria) and FreeCS and UPM were used by Johnson *et al.* [31]. These 14 programs range from 6K to 916K LoC and cover a wide range of functionality (see the description column). In total, the 12 programs on GitHub have a large user base with 5363 watchers, 24 723 stars, and 10 561 forks on GitHub. The other two, Freecs and HSQDB have 41K+ and 1M+ downloads, respectively, on sourceforge.net as of October 2018.

TABLE I
Programs in the real-world benchmark.

program	description	LoC	# reports TP	FP
Apollo-0.9.1	distributed config. [4]	915 602	4	6
BioJava-4.2.8	comp. genomics frmwrk [49]	184 040	26	32
FreeCS-1.2	chat server [13]	27 252	10	0
Giraph-1.1.0	graph processing sys. [15]	120 017	1	8
H2-DB-1.4.196	database engine [21]	235 522	17	30
HSQLDB-2.4.0	database engine [26]	366 902	43	15
Jackrabbit-2.15.7	content repository [25]	416 961	1	6
Jetty-9.4.8	web server w/servlets [28]	650 663	12	4
Joda-Time-2.9.9	date and time frmwrk [30]	277 230	2	3
JPF-8.0	symbolic execution tool [27]	119 186	15	27
MyBatis-3.4.5	persistence frmwrk [45]	133 600	3	15
OkHttp-3.10.0	Android HTTP client [47]	60 774	10	2
UPM-1.14	password management [58]	6358	2	13
Susi.AI-07260c1	artificial intel. API [60]	65 388	47	46
Total		-	194	206

Running FindSecBugs on these programs resulted in more than 400 vulnerability reports. We then labeled the reports by manually reviewing the code[4], resulting in 194 true and 206 false positives as ground-truth. To label a SA report, we first compute the backward call tree from the method that has the reported error line. Then we inspect the code in all callers until either we find a data-flow from an untrusted source (e.g., user input, http request) without any sanity check—indicating a true positive—or we exhaust the call tree without identifying any tainted or unchecked data-flow—indicating a false positive.

我们手动标记了194个正确报告和206个错误报告

Through this review process, we observed that the false positives we found in the real-world benchmark were significantly different from those in the OWASP benchmark programs. The false positives of FindSecBugs usually happen due to one of three scenarios: (1) the tool over-approximates and incorrectly finds an unrealizable flow; (2) the tool fails to recognize that a tainted value becomes untainted along a path, e.g., due to a sanitization routine; or (3) the source that the tool regards as tainted is actually not tainted. In the OWASP benchmark, false positives mostly stem from the first scenario. In our real-world benchmark, we mostly see only the second and third scenarios. This demonstrates the importance of creating a real-world benchmark for our study.

benchmark和现实程序产生的误报原因存在不同：
1）工具发现不可能实现的流
2）工具不能发现污染源实际上被消除
3）污染源实际上没有被污染

IV. Experimental Setup

In this section, we discuss our experimental setup, including the variations of ML algorithms we compared, and how we divide datasets into training and test sets to mimic two different usage scenarios.

Variations of Machine Learning Algorithms. We compared the four families of ML approaches described in Section II. For learning with HEF, we experimented with 9

[4] The first author performed most of the labeling work while other authors verified a random selection of labelings.

TABLE II
BoW, LSTM, and GGNN approaches

applied preparations	approach name
Occurrence feature vec.	*BoW-Occ*
Frequency feature vec.	*BoW-Freq*
T_{cln}	*LSTM-Raw*
$T_{cln} + T_{ans}$	*LSTM-ANS*
$T_{cln} + T_{ans} + T_{aps}$	*LSTM-APS*
$T_{cln} + T_{ans} + T_{aps} + T_{ext}$	*LSTM-Ext*
Kind, operation, and type node vec.	*GGNN-KOT*
KOT + an extracted item	*GGNN-KOTI*
Node Encoding	*GGNN-Enc*

classification algorithms: Naive Bayes, BayesianNet, DecisionTree (J48), Random Forest, MultiLayerPerceptron (MLP), K*, OneR, ZeroR, and support vector machines, with the 15 features described in Section II-A. We used the WEKA [9] implementations of these algorithms.

我们用WEKA实现这些算法

For the other three families of approaches, we experimented with the variations described in Sections II-C, II-D, II-E. Table II lists these variations with their names and data preparation applied for them. For example, the approach *LSTM-Raw* uses T_{cln} transformations alone, while *LSTM-Ext* uses all four transformations. For BoW, we only used DecisionTree (J48) based on its good performance on HEF approaches. We adapted the LSTM implementation designed by Carrier *et al.* [7] and extended the GGNN implementation from Microsoft Research [43].

Application Scenarios. In practice, we envision two scenarios for using ML to classify false positives. First, developers might continuously run static analysis tools on the same set of programs as those programs evolve over time. For example, a group of developers might use static analysis as they develop their code. In this scenario, the models might learn signals that specifically appear in those programs, certain identifiers, API usage, etc. To mimic this scenario, we divide the OWASP and real-world benchmark randomly into training and test sets. Doing so, both training and test sets will have samples from each program in the dataset. We refer to the real-world random split dataset as RW-Rand for short.

以API为单位划分数据集

Second, developers might want to deploy static analysis on a new subject program. In this scenario, the training would be performed on one set of programs, and learned model would be applied to another. To mimic this scenario, we divide the programs randomly so that a collection of programs forms the training set and the remaining ones form the test set. To our knowledge, this scenario has not been studied in the literature for the SA report classification problem. Note that the OWASP benchmark is not appropriate for the second scenario as all the programs in the benchmark were developed by same people and hence share many common properties like variable names, length, API usage, etc. We refer to the real-world program-wise split dataset as RW-PW for short.

以程序为单位划分数据集

Training Configuration. Evaluating ML algorithms requires separating data points into a training set, used to estimate model parameters, and a test set, used to evaluate classifier performance. For both scenarios, we performed 5-fold cross-validation, i.e., 5 random splits for the first scenario and 5 program-wise splits for the second scenario, by dividing the dataset into 5 subsets and using 4 subsets for training 1 subset for testing for each 4-way combinations. Furthermore, we repeat each execution 5 times with different random seeds. The purpose of these many repetitions (5-fold cross-validation × 5 random seeds = 25 runs) is to evaluate whether the results are consistent (see **RQ3**).

LSTM and GGNN are trained using an iterative algorithm which requires users to provide a stopping criterion. We set a timeout of 5 hours and we ended training if there was no accuracy improvement for 20 and 100 epochs, respectively. We made this choice because an LSTM epoch takes about 5 times longer to run than a GGNN epoch, making this threshold approximately the same in terms of clock time.

For the LSTM, we conducted small-scale experiments of 15 epochs with the RW-Rand dataset and *LSTM-Ext* to determine the word embedding dimension for tokens and batch size. We tested $4, 8, 12, 16, 20$, and 50 for the word embeddings and $1, 2, 4, 8, 16$, and 32 for the batch size. We observed that word embedding dimension 8 and batch size 8 led to the highest test accuracy on average and thus we use these values in the remaining experiments. We also used this embedding dimension for the pre-training of *GGNN-Enc*.

Metrics. To evaluate the efficiency of the ML algorithms in terms of time, we use the *training time* and *number of epochs*. After loading a learned model to the memory, the time to test a data point is negligible (around a second) for all ML algorithms. To evaluate effectiveness, we use *recall*, *precision*, and *accuracy* as follows:

$$Precision(P) = \frac{\text{\# of correctly classified true positives}}{\text{\# of samples classified as true positive}}$$

$$Recall(R) = \frac{\text{\# of correctly classified true positives}}{\text{all true positives}}$$

$$Accuracy(A) = \frac{\text{\# of correctly classified samples}}{\text{\# of all samples, i.e., size of test set}}$$

Accuracy is a good indicator of effectiveness for our study because there is no trivial way to achieve high accuracy having an even distribution of samples for each class. Recall can be more useful when missing a true positive report is unacceptable (e.g., when analyzing safety-critical systems). Precision can be more useful when the cost of reviewing false positive report is unacceptable. All three metrics are computed using the test portion of the datasets.

Research questions. With the above experimental setup, we conducted our study to answer the following research questions (RQ).

- **RQ1 (overall performance comparison):** Which family of approaches perform better overall?
- **RQ2 (effect of data preparation):** What is the effect of data preparation on performance?
- **RQ3 (variability analysis):** What is the variability in the results?
- **RQ4 (further interpreting the results):** How do the approaches differ in what they learn?

TABLE III

Recall, precision and accuracy results for the approaches in Table II and four most accurate algorithms for HEF, sorted by accuracy. Numbers in bigger font are median of 25 runs, and numbers in smaller font semi-interquartile range (SIQR). The dashed-lines separate the approaches that have high accuracy from others at a point where there is a relatively large gap.

dataset	approach	recall		precision		accuracy	
OWASP	*LSTM-Raw*	100.00	0	100.00	0	100.00	0
	LSTM-ANS	99.15	0.74	98.74	0.42	99.37	0.42
	LSTM-Ext	98.94	1.90	99.57	0.44	99.16	1.16
	LSTM-APS	98.30	0.42	99.14	0.21	98.53	0.27
	BoW-Occ	97.90	0.45	97.90	1.25	97.47	0.74
	BoW-Freq	97.90	0.45	97.00	0.25	97.26	0.31
	GGNN-Enc	92.00	5.00	94.00	5.25	94.00	1.60
	HEF-J48	88.50	1.65	75.10	0.50	79.96	0.21
	GGNN-KOTI	78.50	6.25	81.00	2.50	79.00	1.95
	HEF-RandomForest	85.50	1.65	74.10	0.65	78.32	0.50
	GGNN-KOT	80.00	3.25	77.50	2.00	78.00	0.95
	*HEF-K**	84.70	2.05	73.60	0.90	77.68	1.37
	HEF-MLP	79.10	7.00	70.90	2.10	73.00	1.27
RW-Rand	*LSTM-Raw*	90.62	2.09	86.49	3.52	89.33	2.19
	LSTM-Ext	90.62	4.41	85.29	3.20	89.04	1.90
	LSTM-APS	91.43	4.02	86.11	3.99	87.67	2.85
	LSTM-ANS	89.29	2.86	84.21	3.97	87.67	1.59
	BoW-Freq	86.10	2.30	87.90	1.85	87.14	1.85
	BoW-Occ	84.40	4.45	87.50	3.85	85.53	2.45
	GGNN-KOTI	83.00	4.50	84.00	3.50	84.21	1.55
	*HEF-K**	80.00	3.95	85.70	2.30	84.00	0.89
	HEF-RandomForest	75.00	1.40	84.40	3.20	84.00	0.93
	GGNN-KOT	89.00	7.00	80.00	7.00	83.56	3.48
	GGNN-Enc	80.00	6.00	78.00	4.50	82.19	3.63
	HEF-J48	78.10	2.15	82.40	0.90	81.33	0.92
	HEF-MLP	71.40	2.80	86.20	6.10	81.33	1.97
RW-PW	*LSTM-Ext*	78.57	12.02	76.19	5.20	80.00	4.00
	LSTM-APS	70.27	14.59	76.47	6.70	78.48	3.33
	LSTM-ANS	62.16	25.58	75.76	7.02	74.68	3.85
	LSTM-Raw	67.57	31.91	79.66	8.40	74.67	4.08
	GGNN-Enc	77.00	36.00	75.00	19.50	74.67	5.89
	GGNN-KOT	77.00	29.50	72.00	16.25	74.00	5.84
	HEF-MLP	58.10	14.65	70.40	9.40	73.08	7.76
	GGNN-KOTI	65.00	33.50	75.00	11.00	72.02	5.12
	*HEF-K**	66.10	24.50	60.60	14.90	68.00	9.75
	HEF-J48	60.70	11.65	72.70	12.80	65.33	8.04
	HEF-RandomForest	62.50	24.30	60.30	5.55	63.44	2.67
	BoW-Occ	50.00	12.90	65.00	22.30	51.32	4.61
	BoW-Freq	47.80	16.50	65.70	14.70	51.25	8.55

TABLE IV

Number of epochs and training times for the LSTM and GGNN approaches. Median and SQIR values as in Table III

		# of epochs		training time(min)	
OWASP	*LSTM-Raw*	170	48	23	11
	LSTM-ANS	221	47	32	4
	LSTM-APS	237	35	31	4
	LSTM-Ext	197	79	37	20
	GGNN-KOT	303	113	28	10
	GGNN-KOTI	218	62	20	6
	GGNN-Enc	587	182	54	17
RW-Rand	*LSTM-Raw*	62	1	303	1
	LSTM-ANS	64	1	303	1
	LSTM-APS	63	1	303	1
	LSTM-Ext	50	0	304	2
	GGNN-KOT	325	6	301	0
	GGNN-KOTI	325	6	300	0
	GGNN-Enc	326	4	300	0
RW-PW	*LSTM-Raw*	63	2	301	2
	LSTM-ANS	65	2	301	6
	LSTM-APS	65	2	302	2
	LSTM-Ext	52	2	303	2
	GGNN-KOT	284	54	250	47
	GGNN-KOTI	215	21	194	17
	GGNN-Enc	245	50	211	58

All experiments were carried on a 64-bit Linux (version 3.10.0-693.17.1.el7) VM running on 12-core Intel Xeon E312xx 2.4GHz (Sandy Bridge) processor and 262GB RAM.

V. Analysis of Results

In total, we trained 1350 SA report classification models: 3 datasets × 5 splits × 5 random seeds × (9 algorithms for HEF + 2 BoW variations + 4 data preparation routines for LSTM + 3 node representations for GGNN). The summary of the results can be found in Tables III and IV, as the median and semi-interquartile range (SIQR) of 25 runs. We report median and SIQR because we do not have any hypothesis about the underlying distribution of the data. Note that, for HEF, we list the four algorithms that had the best accuracy: K*, J48, RandomForest, and MLP. We now answer each RQ.

A. RQ1: Overall Performance Comparison

In this section, we analyze the overall performance of four main learning approaches using the accuracy metric and we observe that the trends we mention here also hold for the recall and precision metrics. In Table III, we separated high performing approaches from others with a dashed-line at points where there is a large gap in accuracy. Overall, *LSTM-based approaches outperform other learning approaches* in accuracy. The deep learning approaches (LSTM and GGNN) classify false positives more accurately than HEF and BoW, at the cost of longer training times. The gap between LSTM and GGNN and other approaches is larger in the second application scenario suggesting that the hidden representations learned generalize across programs better than HEF and BoW features. Next, we analyze the results for each dataset.

For the OWASP dataset, all LSTM approaches achieve above 98% for recall, precision, and accuracy metrics. BoW approaches are close, achieving about 97% accuracy. The HEF approaches, however, are all below the dashed-line with below 80% accuracy. We conjecture that the features used by HEF do not adequately capture the symptoms of false (or true) positive reports (see Section V-D). The GGNN variations have a big difference in accuracy. The *GGNN-Enc* achieves 94%, while the other two variations achieve around 80% accuracy. This suggests that for the OWASP dataset, the value of the PDG nodes, i.e., the textual content of the programs, carry useful signals to be learned during training. This also explains the outstanding performance of the BoW and LSTM approaches, as they mainly use this textual content in training.

For the RW-Rand dataset, two LSTM approaches achieve close to 90% accuracy, followed by BoW approaches at around 86%. GGNN and HEF approaches achieve around 80% accuracy. This result suggests that the RW-Rand dataset contains more relevant features the HEF approaches can take advantage of, and we conjecture that the overall accuracy of the other three algorithms dropped because of the larger programs and vocabulary in this dataset. Table V shows the number of the words and length of samples for the LSTM approaches (the

TABLE V
DATASET STATS FOR THE LSTM APPROACHES. FOR THE SAMPLE LENGTH, NUMBERS IN THE NORMAL FONT ARE THE MAXIMUM AND IN THE SMALLER FONT ARE THE MEAN.

	dictionary size		sample length	
approach	OWASP	real-world	OWASP	real-world
LSTM-Raw	333	13 237	735 224	156 393 18524
LSTM-ANS	284	9724	706 212	149 886 18104
LSTM-APS	284	9666	706 212	150 755 18378
LSTM-Ext	251	4730	925 277	190 950 23031

normal font is the maximum while the smaller font is the mean). As expected, the dictionary gets smaller while the samples get larger as we apply more data preparation. For GGNN, the number of nodes is 24 on average and 82 at most, the number of edges is 47 on average and 174 at most in the OWASP dataset. The real-world dataset has 1880 average to 16 479 maximum nodes, and 6411 average to 146 444 maximum edges. The real-world dataset is significantly larger both in dictionary sizes and sample lengths.

For the RW-PW dataset, all the accuracy results except LSTM-Ext are below 80%. Recall that this split was created for the second application scenario where the training is performed using one set of programs and testing is done using others. We observe the neural networks (i.e., LSTM and GGNN) still produce reasonable results, while the results of HEF and BoW dropped significantly. This suggests that neither the hand-engineered features nor the textual content of the programs are adequate for the second application scenario, without learning any structural information from the programs.

Next, Both HEF and BoW approaches are very efficient. All their variations completed training in less than a minute for all datasets, while the LSTM and GGNN approaches run for hours for the RW-Rand and RW-PW datasets (Table IV). This is mainly due to the large number of parameters being optimized in the LSTM and GGNN.

Lastly, note that the results on the OWASP dataset (Table III) are directly comparable with the results reported by Koc *et al.* [35], which report 85% and 90% accuracy for program slice and control-flow graph representations, respectively. In this paper, we only experimented with program slices as they are a precise summarization of the programs. With the same dataset, our *LSTM-Ext* approach, which does not learn from any program-specific tokens, achieves 99.57% accuracy. Therefore, we conjecture these improvements are due to the better and more precise data preparation routines we perform.

B. RQ2: Effect of Data Preparation

We now analyze the effect of different data preparation techniques for the ML approaches. Recall the goal of data preparation is to provide the most effective use of information that is available in the program context. *We found LSTM-Ext produced the overall best accuracy results across the three datasets. The different node representations of GGNN present tradeoffs, while the BoW variations produced similar results.*

Four code transformation routines were introduced for LSTM. *LSTM-Raw* achieves 100% accuracy on the OWASP dataset. This is because *LSTM-Raw* performs only basic data cleansing and tokenization, with no abstraction for variable, method, and class identifiers. Many programs in the OWASP benchmark have variables named "safe," "unsafe," "tainted," etc., giving away the answer to the classification task. On the other hand, the RW-PW dataset benefits from more transformation routines that perform abstraction and word extraction. *LSTM-Ext* outperformed *LSTM-Raw* by 5.33% in accuracy for the RW-PW dataset.

We presented three node representation techniques for GGNN. For the OWASP dataset, we observe a significant improvement in accuracy from 78% with *GGNN-KOT* to 94% with *GGNN-Enc*. This suggests that very basic structural information from the OWASP programs (i.e., the kind, operation, and type information included in *GGNN-KOT*) carries limited signal about true and false positives, while the textual information included in *GGNN-Enc* carries more signal, leading to a large improvement. This trend, however, is not preserved on the real-world datasets. All GGNN variations (*GGNN-KOT*, *GGNN-KOTI*, and *GGNN-Enc*) performed similarly with 83.56%, 84.21%, and 82.19% accuracy, respectively, on the RW-Rand, and 74%, 72%, and 74.67% accuracy on the RW-PW datasets. Overall, we think the GGNN trends are not clear partly because of the nature of data such as sample lengths, dictionary and dataset sizes (Tables I and V). Moreover, the information encoded in the *GGNN-KOT* and *GGNN-KOTI* approaches is very limited whereas it might be too much condensed in *GGNN-Enc* (taking the average over the embeddings of all tokens that appear in the statement), making the signals harder to learn.

BoW-Occ and *BoW-Freq* had similar accuracy in general. The largest difference is the 85.53% and 87.14% accuracy for *BoW-Occ* and *BoW-Freq*, respectively, on the RW-Rand dataset. This result suggests that checking the presence of a word is almost as useful as counting its occurrences.

C. RQ3: Variability Analysis

In this section, we analyze the variance in the recall, precision, and accuracy results using the semi-interquartile range (SIQR) value given in the smaller font in Table III.

Note that, unlike other algorithms, J48 and K* deterministically produce the same models when trained on the same training set. The variance observed for J48 and K* is only due to the different splits of the same dataset.

On the OWASP dataset, all approaches have little variance, except for a 7% SIQR for the recall value of HEF-MLP.

On the RW-Rand dataset, SIQR values are relatively higher for all approaches but still under 4% for many of the high performing approaches. The *BoW-Freq* approach has the minimum variance for recall, precision, and accuracy. The *LSTM-ANS* and *LSTM-Ext* follow this minimum variance result. And the HEF-based approaches lead to the highest variance overall.

On the RW-PW dataset, the variance is even bigger. For recall in particular, we observe SIQR values around 30% with some of the HEF, LSTM, and GGNN approaches. The best performing two LSTM approaches, *LSTM-Ext* and *LSTM-APS*, have less than 4% difference between quartiles in accuracy. We

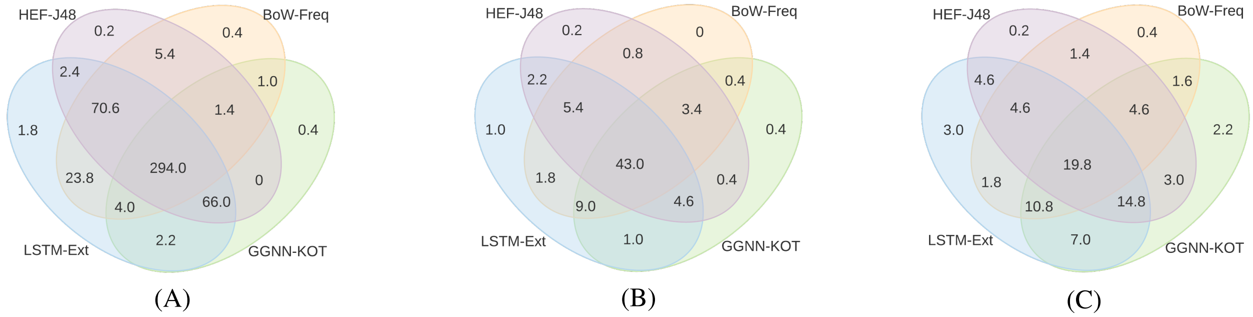


Fig. 2. Venn diagrams of the number of correctly classified examples for *HEF-J48*, *BoW-Freq*, *LSTM-Ext*, and *GGNN-KOT* approaches, average for 5 models trained for the OWASP (A), RW-Rand (B), and RW-PW (C) datasets (474, 74, and 80 test samples respectively).

conjecture this is because the accuracy value directly relates to the loss function being optimized (minimized), while recall and precision are indirectly related. Lastly, applying more data preparation for LSTM leads to a smaller variance for all the three metrics for the PW-RW dataset.

D. RQ4: Further Interpreting the Results

To draw more insights on the above results, we further analyze four representative variations, one in each family of approaches. We chose *HEF-J48*, *BoW-Freq*, *LSTM-Ext*, and *GGNN-KOT* because these instances generally produce the best results in their family. Figure 2 shows Venn diagrams that illustrate the distribution of the correctly classified reports, for these approaches with their overlaps (intersections) and differences (as the mean for 5 models). For example, in Figure 2-A, the value 294 in the region covered by all four colors means these reports were correctly classified by all four approaches, while the value 1.8 in the blue only region mean these reports were correctly classified only by LSTM.

The RW-Rand results in Figure 2-B show that 43 reports were correctly classified by all four approaches, meaning these reports have symptoms that are detectable by all approaches. On the other hand, 30.6 (41%) of the reports were misclassified by at least one approach.

The RW-PW results in Figure 2-C show that only 20 reports were correctly classified by all approaches, which is mostly due to the poor performance of the *HEF-J48* and *BoW-Freq*. The *LSTM-Ext* and *GGNN-KOT* can correctly classify about 10 more reports which were misclassified both by the *HEF-J48* and *BoW-Freq*. This suggests that the *LSTM-Ext* and *GGNN-KOT* captured more generic signals that hold across programs.

Last, the overall results in Figure 2 show that no single approach correctly classified a superset of any other approach, and therefore there is a potential for achieving better accuracy by combining multiple approaches.

Figure 3-A shows a sample program from the OWASP dataset to demonstrate the potential advantage of the *LSTM-Ext*. At line 2, the param variable receives a value from request.getQueryString(). This value is tainted because it comes from the outside source HttpServletRequest. The switch block on lines 7 to 16 controls the value of the variable bar. Because switchTarget is assigned ‘B’ on line 4, bar always receives the value "bob". On line 17, the variable sql is assigned to a string containing bar, and then used as a parameter in the statement.executeUpdate(sql) call on line 20. In this case, FindSecBugs overly approximates that the tainted value read into the param variable might reach the executeUpdate statement, which would be a potential SQL injection vulnerability, and thus generates a vulnerability warning. However, because bar always receives the safe value "bob", this report is a false positive.

Among the four approaches we discuss here, this report was correctly classified only by *LSTM-Ext*. To illustrate the reason, we show the different inputs of these approaches. Figure 3-B shows the sequential representation used by *LSTM-Ext*. *HEF-J48* used the following feature vector:

$$\begin{gathered}[\boldsymbol{rule\_name} : SQL\_INJECTION, \\ \boldsymbol{sink\_line} : 19, \boldsymbol{sink\_identifier} : Statement.executeUpdate, \\ \boldsymbol{source\_line} : 2, \boldsymbol{source\_identifier} : request.getQueryString, \\ \boldsymbol{functions} : 4, \boldsymbol{witness\_length} : 2, \boldsymbol{number\_bugs} : 1, \boldsymbol{conditions} : 1, \\ \boldsymbol{severity} : 5, \boldsymbol{confidence} : High, \boldsymbol{time} : 2, \boldsymbol{classes\_involved} : 1]\end{gathered}$$

Notice that this feature vector does not include any information about the string variable guess, the switch block, or overall logic that exists in the program. Instead, it relies on correlations that might exist for the features above. For this example, such correlations weight more for the true positive decision, thus lead to a misclassification.

On the other hand, the *LSTM-Ext* representation includes the program information. For example, VAR 6 gets assigned to the return value of the request.getQueryString method, and VAR 10 is defined as STR 1 . char At (1) (STR 1 is the first string that appears in this program, i.e., "ABC"). We see the tokens switch VAR 10 at line 5 corresponding to the switch statement. Then, we see string and SQL operations through lines 7 to 12, followed by a PHI instruction at line 13. This sequential representation helps *LSTM-Ext* to correctly classify the example as a false positive.

Last, *BoW-Freq* misclassified this example using the tokens in Figure 3-B without their order. This suggests that the overall correlation of the tokens that appear in this slice does not favor the false positive class. We argue that the correct classification by *LSTM-Ext* was not due to the presence of certain tokens, but rather due to the sequential structure.

```
public void doPost(HttpServletRequest request...){
  String param = request.getQueryString();
  String sql, bar, guess = "ABC";
  char switchTarget = guess.charAt(1); // 'B'
  // Assigns param to bar on conditions 'A' or 'C'
  switch (switchTarget) {
    case 'A':
      bar = param; break;
    case 'B': // always holds
      bar = "bob"; break;
    case 'C':
      bar = param; break;
    default:
      bar = "bob's your uncle"; break;
  }
  sql = "UPDATE USERS SET PASSWORD='" + bar + "'
        WHERE USERNAME='foo'";
  try {
    java.sql.Statement statement =
          DatabaseHelper.getSqlStatement();
    int count = statement.executeUpdate(sql);
  } catch (java.sql.SQLException e) {
    throw new ServletException(e);
  }}
```

(A)

```
org owasp benchmark UNK UNK do Post ( Http Servlet
Request Http Servlet Response ) : String VAR 6 = p
1 request get Query String ( ) : C VAR 10 = STR 1
char At ( 1 ) : switch VAR 10 : String Builder VAR
14 = new String Builder : String Builder VAR 18 =
VAR 14 append ( STR 0 ) : String Builder VAR 20 =
VAR 18 append ( VAR 13 ) : String Builder VAR 23 =
VAR 20 append ( STR 3) : String VAR 25 = VAR 23 to
String ( ) : java sql Statement VAR 27 = get Sql
Statement ( ) : I VAR 29 = VAR 27 execute Update (
VAR 25 ) : PHI VAR 13 = VAR 6 STR 4 VAR 6 STR 2
```

(B)

Fig. 3. An example program (simplified) from the OWASP benchmark that was correctly classified only by *LSTM-Ext* (A) and the sequential representation used for *LSTM-Ext* (B)

E. Threats To Validity

There are several threats to the validity of our study. First, the benchmarks may not be representative. Indeed, the OWASP benchmark is synthetic. Therefore, we collected the first real-world benchmark for classifying SA results, consisting of 14 programs to increase the generalizability of our results. In addition, our real-world benchmark consists of 400 data points which may not be large enough to train neural networks with high confidence. We repeated the experiments using different random seeds and data splittings to analyze the variability that might be caused by having limited data. Second, we ran our experiments on a virtual machine, which may affect the training times. However, since these models would be trained offline and very rarely, we are primarily interested in effectiveness of the approaches in this study.

VI. Related Work

Two lines of research are most related to the work described in this paper. First, we briefly discuss research that uses ML to classify false positive SA reports. Second, we briefly discuss the broader use of ML, specifically, NLP approaches, directly on the source code. To the best of our knowledge, there are no prior empirical studies of ML approaches for false positive SA report classification.

False positive report filtering using ML. To date, most research aimed at filtering false positive SA reports has used hand-engineered features [22], [23], [61], [69]. For instance, Tripp *et al.* [61] identify 14 such features for false positive XSS reports generated for JavaScript programs. We evaluated this approach by adopting these 14 features for Java programs, attempting to hew closely to the type of features used in the original work. More recently, Koc *et al.* [35] conducted a case study applying a recurrent neural network approach to the synthetic OWASP benchmark. Although the results were promising, the approach had not been applied to real-world programs. We extend and evaluate this approach with more precise program summarization and data preparation routines using real-world programs. Raghothaman *et al.* [50] build on an ML approach, Bayesian inference, that additional relies on direct feedback from human tool users. We did not include this work in our evaluation because it works on per program basis and requires user input. Furthermore, none of the existing work in this line of research has studied our second application scenario which tries to generalize learning to new programs.

NLP techniques applied to code. Multiple researchers have successfully applied NLP techniques to programs to tackle SE problems such as clone detection [66], API mining [12], [19], variable naming and renaming [1], [51], code suggestion and completion [46], [52], [62], bug detection [3]. Allamanis *et al.* [2] conducted an extensive survey of such research efforts. However, none of these efforts addresses the problem of identifying and distinguishing false positive SA reports.

VII. Conclusion and Future Work

We presented the first empirical study that evaluates four families of ML approaches, i.e, HEF, BoW, LSTM, and GGNN, for classifying static analysis results to filter false positives from true positives. To adapt these approaches to classify false positives, we introduced new code transformations for preparing data as inputs,

In our experiments, we compared 13 ML approaches from four families, using two benchmarks under two application scenarios. The results of our experiments suggest that the LSTM approach generally achieves better accuracy. We also observed that, across all approaches, the second application scenario in which the training is done with one set of programs and the models are tested on other programs is more challenging, It requires learning the symptoms of true/false positive reports that holds across programs. Particularly in this application scenario, we observed that more detailed data preparation with abstraction and word extraction leads to significant increases in accuracy. We also showed that there can be higher variance in recall and precision than in accuracy. We conjecture this is because the recall and precision are not directly related to the loss function being optimized in training.

In future work, we plan to explore a voting scheme that combines different ML approaches to create an ensemble classifier that can achieve better accuracy. In addition, we will extend the experiments with other SA tools, which also requires collecting new ground-truth datasets.

Acknowledgment

This work was sponsored by the Department of Homeland Security under grant #D15PC00169.

References

[1] Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. Suggesting Accurate Method and Class Names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2015), ESEC/FSE 2015, ACM, pp. 38–49.

[2] Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. A survey of machine learning for big code and naturalness. *ACM Comput. Surv. 51*, 4 (July 2018), 81:1–81:37.

[3] Allamanis, M., Brockschmidt, M., and Khademi, M. Learning to Represent Programs with Graphs. *arXiv:1711.00740 [cs]* (Nov. 2017).

[4] Apollo: a distributed configuration center. https://github.com/ctripcorp/apollo, 2018.

[5] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications* (New York, NY, USA, 2006), OOPSLA '06, ACM, pp. 169–190.

[6] Burato, E., Ferrara, P., and Spoto, F. Security analysis of the owasp benchmark with julia. In *Proc. of ITASEC17, the rst Italian Conference on Security, Venice, Italy* (2017).

[7] Carrier, P.-L., and Cho, K. LSTM Networks for Sentiment Analysis: DeepLearning 0.1 documentation. http://deeplearning.net/tutorial/lstm.html.

[8] Dam, H. K., Tran, T., and Pham, T. T. M. A deep language model for software code. In *FSE 2016: Proceedings of the Foundations Software Engineering International Symposium* (2016), pp. 1–4.

[9] Eibe, F., Hall, M., and Witten, I. The weka workbench. *Morgan Kaufmann* (2016).

[10] Erez, G., Yahav, E., and Sagiv, M. *Generating concrete counterexamples for sound abstract interpretation.* Citeseer, 2004.

[11] Find Security Bugs, version 1.4.6. http://find-sec-bugs.github.io, Accessed: 2018-10-02.

[12] Fowkes, J., and Sutton, C. Parameter-free Probabilistic API Mining Across GitHub. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2016), FSE 2016, ACM, pp. 254–265.

[13] Free chat-server: A chatserver written in java. https://sourceforge.net/projects/freecs.

[14] GERS, F. A., SCHMIDHUBER, J., and CUMMINS, F. Learning to forget: Continual prediction with lstm. *Neural computation 12*, 10 (2000), 2451–2471.

[15] Giraph : Large-scale graph processing on hadoop. http://giraph.apache.org.

[16] Goldberg, Y. Neural network methods for natural language processing. *Synthesis Lectures on Human Language Technologies 10*, 1 (2017), 1–309.

[17] Goldberg, Y., and Levy, O. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv:1402.3722* (2014).

[18] Gori, M., Monfardini, G., and Scarselli, F. A new model for learning in graph domains. In *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on* (2005), vol. 2, IEEE, pp. 729–734.

[19] Gu, X., Zhang, H., Zhang, D., and Kim, S. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2016), ACM, pp. 631–642.

[20] Gulavani, B. S., and Rajamani, S. K. Counterexample Driven Refinement for Abstract Interpretation. In *Tools and Algorithms for the Construction and Analysis of Systems* (Mar. 2006), Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 474–488.

[21] H2 database engine. http://www.h2database.com.

[22] Heckman, S. S. Adaptive probabilistic model for ranking code-based static analysis alerts. In *Software Engineering - Companion, 2007. ICSE 2007 Companion. 29th International Conference on* (May 2007), pp. 89–90.

[23] Heckman, S. S. *A systematic model building process for predicting actionable static analysis alerts.* North Carolina State University, 2009.

[24] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural Computation 9*, 8 (Nov. 1997), 1735.

[25] Hypersql database. http://hsqldb.org.

[26] Apache jackrabbit is a fully conforming implementation of the content repository for java technology api. http://jackrabbit.apache.org.

[27] Java pathfinder. https://github.com/javapathfinder.

[28] Jetty: lightweight highly scalable java based web server and servlet engine. https://www.eclipse.org/jetty, 2018.

[29] Joana (java object-sensitive analysis) - information flow control framework for java. https://pp.ipd.kit.edu/projects/joana.

[30] Joda-time a quality replacement for the java date and time classes. http://www.joda.org/joda-time.

[31] Johnson, A., Waye, L., Moore, S., and Chong, S. Exploring and Enforcing Security Guarantees via Program Dependence Graphs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2015), PLDI '15, ACM, pp. 291–302.

[32] Johnson, B., Song, Y., Murphy-Hill, E., and Bowdidge, R. Why Don't Software Developers Use Static Analysis Tools to Find Bugs? In *Proceedings of the 2013 International Conference on Software Engineering* (Piscataway, NJ, USA, 2013), ICSE '13, IEEE Press, pp. 672–681.

[33] Jung, Y., Kim, J., Shin, J., and Yi, K. Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis. In *Static Analysis* (Sept. 2005), Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 203–217.

[34] Kim, Y., Lee, J., Han, H., and Choe, K.-M. Filtering false alarms of buffer overflow analysis using SMT solvers. *Information and Software Technology 52*, 2 (Feb. 2010), 210–219.

[35] Koc, U., Saadatpanah, P., Foster, J. S., and Porter, A. A. Learning a classifier for false positive error reports emitted by static code analysis tools. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (New York, NY, USA, 2017), MAPL 2017, ACM, pp. 35–42.

[36] Kremenek, T., and Engler, D. Z-ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *Proceedings of the 10th International Conference on Static Analysis* (Berlin, Heidelberg, 2003), SAS'03, Springer-Verlag, pp. 295–315.

[37] Kushman, N., and Barzilay, R. Using semantic unification to generate regular expressions from natural language. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (2013), pp. 826–836.

[38] Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. Gated Graph Sequence Neural Networks. *arXiv:1511.05493* (Nov. 2015).

[39] Ling, W., Blunsom, P., Grefenstette, E., Hermann, K. M., Kočiskỳ, T., Wang, F., and Senior, A. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (2016), vol. 1, pp. 599–609.

[40] Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J. N., Chang, B.-Y. E., Guyer, S. Z., Khedker, U. P., Møller, A., and Vardoulakis, D. In Defense of Soundiness: A Manifesto. *Communications of the ACM 58*, 2 (2015), 44–46.

[41] Lukins, S. K., Kraft, N. A., and Etzkorn, L. H. Bug localization using latent Dirichlet allocation. *Information and Software Technology 52*, 9 (2010), 972 – 990.

[42] Mandic, D. P., and Chambers, J. *Recurrent neural networks for prediction: learning algorithms, architectures and stability*. John Wiley & Sons, Inc., 2001.

[43] Microsoft gated graph neural networks. https://github.com/Microsoft/gated-graph-neural-network-samples.

[44] Mikolov, T., Chen, K., Corrado, G., Dean, J., Sutskever, L., and Zweig, G. word2vec. *https://code.google.com/p/word2vec* (2013).

[45] Mybatis: Sql mapper framework for java. http://www.mybatis.org/mybatis-3.

[46] Nguyen, T. T., Nguyen, A. T., Nguyen, H. A., and Nguyen, T. N. A Statistical Semantic Language Model for Source Code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2013), ESEC/FSE 2013, ACM, pp. 532–542.

[47] Okhttp: An http & http/2 client for android and java applications. http://square.github.io/okhttp.

[48] The OWASP Benchmark for Security Automation, version 1.1, 2014. https://www.owasp.org/index.php/Benchmark, Accessed: 2018-01-04.

[49] Prlić, A., Yates, A., Bliven, S. E., Rose, P. W., Jacobsen, J., Troshin, P. V., Chapman, M., Gao, J., Koh, C. H., Foisy, S., et al. Biojava: an open-source framework for bioinformatics in 2012. *Bioinformatics 28*, 20 (2012), 2693–2695.

[50] Raghothaman, M., Kulkarni, S., Heo, K., and Naik, M. User-guided program reasoning using bayesian inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2018), PLDI 2018, ACM, pp. 722–735.

[51] Raychev, V., Vechev, M., and Krause, A. Predicting Program Properties from "Big Code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2015), POPL '15, ACM, pp. 111–124.

[52] Raychev, V., Vechev, M., and Yahav, E. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, ACM, pp. 419–428.

[53] Rival, X. Understanding the origin of alarms in astrÉe. In *Proceedings of the 12th International Conference on Static Analysis* (Berlin, Heidelberg, 2005), SAS'05, Springer-Verlag, pp. 303–319.

[54] Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1988), POPL '88, ACM, pp. 12–27.

[55] Russell, S. J., and Norvig, P. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[56] Sak, H., Senior, A., and Beaufays, F. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Fifteenth annual conference of the international speech communication association* (2014).

[57] Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. The Graph Neural Network Model. *IEEE Transactions on Neural Networks 20*, 1 (Jan. 2009), 61–80.

[58] Smith, A. Universal password manager. http://upm.sourceforge.net.

[59] Sureka, A., and Jalote, P. Detecting Duplicate Bug Report Using Character N-Gram-Based Features. In *2010 Asia Pacific Software Engineering Conference* (Nov. 2010), pp. 366–374.

[60] api.susi.ai - Software and Rules for Personal Assistants. http://susi.ai.

[61] Tripp, O., Guarnieri, S., Pistoia, M., and Aravkin, A. ALETHEIA: Improving the Usability of Static Security Analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2014), CCS '14, ACM, pp. 762–774.

[62] Tu, Z., Su, Z., and Devanbu, P. On the Localness of Software. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), FSE 2014, ACM, pp. 269–280.

[63] T. J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/.

[64] Wang, J., Wang, S., and Wang, Q. Is there a "golden" feature set for static warning identification?: An experimental evaluation. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (New York, NY, USA, 2018), ESEM '18, ACM, pp. 17:1–17:10.

[65] Weiser, M. Program slicing. In *Proceedings of the 5th international conference on Software engineering* (1981), IEEE Press, pp. 439–449.

[66] White, M., Tufano, M., Vendome, C., and Poshyvanyk, D. Deep Learning Code Fragments for Code Clone Detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2016), ASE 2016, ACM, pp. 87–98.

[67] Xypolytos, A., Xu, H., Vieira, B., and Ali-Eldin, A. M. A framework for combining and ranking static analysis tool findings based on tool performance statistics. In *Software Quality, Reliability and Security Companion (QRS-C), 2017 IEEE International Conference on* (2017), IEEE, pp. 595–596.

[68] Ye, X., Shen, H., Ma, X., Bunescu, R., and Liu, C. From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering. In *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA, 2016), ICSE '16, ACM, pp. 404–415.

[69] Yüksel, U., and Sözer, H. Automated Classification of Static Code Analysis Alerts: A Case Study. In *2013 IEEE International Conference on Software Maintenance* (Sept. 2013), pp. 532–535.

[70] Zhang, X., Si, X., and Naik, M. Combining the Logical and the Probabilistic in Program Analysis. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (New York, NY, USA, 2017), MAPL 2017, ACM, pp. 27–34.