Empirical Software Engineering

An Empirical Assessment of Machine Learning Approaches for Triaging Reports of Static Analysis Tools

--Manuscript Draft--

Manuscript Number:	EMSE-D-22-00091	
Full Title:	An Empirical Assessment of Machine Learning Approaches for Triaging Reports of Static Analysis Tools	
Article Type:	Research Papers	
Keywords:	static analysis; false positive classification; machine learning; deep learning; neural networks; abstract syntax trees	
Corresponding Author:	Sai S Yerramreddy, Ph. D. University of Maryland at College Park College Park, Maryland UNITED STATES	
Corresponding Author Secondary Information:		
Corresponding Author's Institution:	University of Maryland at College Park	
Corresponding Author's Secondary Institution:		
First Author:	Sai S. Yerramreddy	
First Author Secondary Information:		
Order of Authors:	Sai S. Yerramreddy	
	Austin Mordahl	
	Ugur Koc	
	Shiyi Wei	
	Jeffrey S. Foster	
	Marine Carpuat	
	Adam A. Porter	
Order of Authors Secondary Information:		
Funding Information:	National Science Foundation (CCF-2007314)	Dr. Adam A. Porter
	National Science Foundation (CCF-2008905)	Dr. Shiyi Wei
	National Science Foundation (CCF-2047682)	Dr. Shiyi Wei
	University of Texas at Dallas (Eugene McDermott Graduate Fellowship 202006)	Mr. Austin Mordahl

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

An Empirical Assessment of Machine Learning Approaches for Triaging Reports of Static Analysis Tools

Sai Yerramreddy[1*] · **Austin Mordahl**[2*] · **Ugur Koc**[1] · **Shiyi Wei**[2] · **Jeffrey S. Foster**[3] · **Marine Carpuat**[1] · **Adam A. Porter**[1]

the date of receipt and acceptance should be inserted later

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 research and 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 true results in three static analysis tools (FindSecBugs, CBMC, and JBMC) using multiple ground-truth program sets. We also introduce and evaluate new data preparation routines for recurrent neural networks (RNNs) and node representations for graph neural networks (GNNs). We find that overall classification accuracy reaches a high of 80%-100% for different datasets and application scenarios. We observe that data preparation routines have a positive impact on classification accuracy, with an improvement of up to 2% for RNNs and 7.6% for GNNs. Overall, our results suggest that neural networks (RNNs or GNNs) which learn over a program's source code outperformed 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 for static analysis tools in practice.

Keywords static analysis, false positive classification, machine learning, deep learning, neural networks, abstract syntax trees

*The first two authors contributed equally to this research.

[1]Department of Computer Science, University of Maryland, College Park, USA
E-mail: {saiyr, ukoc, marine, aporter}@cs.umd.edu
[2]Department of Computer Science, University of Texas, Dallas, USA
E-mail: {austin.mordahl, swei}@utdallas.edu
[3]Department of Computer Science, Tufts University, Medford, MA, USA
E-mail: jfoster@cs.tufts.edu

Declarations

Funding: This work was partly supported by NSF grants CCF-2007314, CCF-2008905 and CCF-2047682, the NSF graduate research fellowship program, and Eugene McDermott Graduate Fellowship 202006.

Financial Interests: Dr. Ugur Koc is currently employed by Amazon. Dr. Adam A. Porter receives a salary from association Fraunhofer USA CMA, where he is the Executive and Scientific Director.

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

An Empirical Assessment of Machine Learning Approaches for Triaging Reports of Static Analysis Tools

Sai Yerramreddy[1*] · Austin Mordahl[2*] · Ugur Koc[1] · Shiyi Wei[2] · Jeffrey S. Foster[3] · Marine Carpuat[1] · Adam A. Porter[1]

the date of receipt and acceptance should be inserted later

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 research and 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 true results in three static analysis tools (FindSecBugs, CBMC, and JBMC) using multiple ground-truth program sets. We also introduce and evaluate new data preparation routines for recurrent neural networks (RNNs) and node representations for graph neural networks (GNNs). We find that overall classification accuracy reaches a high of 80%-100% for different datasets and application scenarios. We observe that data preparation routines have a positive impact on classification accuracy, with an improvement of up to 2% for RNNs and 7.6% for GNNs. Overall, our results suggest that neural networks (RNNs or GNNs) which learn over a program's source code outperformed 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 for static analysis tools in practice.

Keywords static analysis, false positive classification, machine learning, deep learning, neural networks, abstract syntax trees

*The first two authors contributed equally to this research.

[1]Department of Computer Science, University of Maryland, College Park, USA
E-mail: {saiyr, ukoc, marine, aporter}@cs.umd.edu
[2]Department of Computer Science, University of Texas, Dallas, USA
E-mail: {austin.mordahl, swei}@utdallas.edu
[3]Department of Computer Science, Tufts University, Medford, MA, USA
E-mail: jfoster@cs.tufts.edu

1 Introduction

Static analysis (SA) tools are designed to detect errors that might jeopardize the correctness, security, and performance of software applications. Unfortunately, SA tools frequently generate large numbers of false results. These can be false positives (i.e., non-errors incorrectly labeled as errors), or false negatives (i.e., real errors that are not detected by the tool). Because of these false results, developers must spend a lot of time manually resolving each SA report, only to potentially miss errors in the target program. Many regard this issue as a key barrier to using SA tools in practice (Johnson et al., 2013). That is, developers perceive the cost of missing some potential errors as much lower than the cost of painstakingly analyzing an incomplete list of hundreds or even thousands of error reports that ultimately turn out be false.

Recently, several researchers have proposed using ML techniques with hand engineered features to classify and filter false positives (Heckman, 2007, 2009; Tripp et al., 2014; Yüksel and Sözer, 2013). 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 (Heckman, 2009; Tripp et al., 2014; Yüksel and Sözer, 2013). 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. (2017) 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 such algorithms are likely to be routinely applied to this use case. First and foremost, there has been relatively little 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 a 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, we describe a systematic, comparative study of multiple ML approaches– hand-engineered features, bag-of-words, recurrent neural networks and graph neural networks– for classifying SA results from three real-world tools: FindSecBugs (Arteau et al., 2018), CBMC (Kroening and Tautschnig, 2014), and JBMC (Cordeiro et al., 2018). In our study, we constructed multiple ground-truth datasets, containing both real-world and artificial benchmarks. The real-world dataset we constructed was for FindSecBugs. This dataset contains reports from 14 Java programs covering a wide range of application domains and 400 vulnerability reports that we manually classified. For FindSecBugs, we also used the OWASP dataset, a popular artificial corpus for evaluating static analysis tools (OWASP, 2014). For CBMC and JBMC, we used the well-known SV-COMP dataset, which aggregates benchmark cases from various contributors into a single corpus (Beyer, 2018, 2019). Our SV-COMP dataset consisted of 1000 C programs and 368 Java programs, including both safe and unsafe programs according to a variety of program verification properties (Section 2).

We introduce key data preparation routines for applying ML approaches to this problem. For the hand-engineered features approach, we wrote simple static analyses for C and Java to extract features such as the number of if statements or number of variables from compiled code. For the three other approaches, we performed preprocessing to give some structural information about the benchmark program to the ML models. Since FindSecBugs emits reports in the forms of flows from sources to sinks, for that tool we computed backwards slices from the reported sink and used the corresponding program dependence graph (PDG). For CBMC and JBMC, which may not emit a specific line we can slice from, we instead generated the abstract syntax tree (AST) of the program. Given these different structures, we processed each model differently: transforming them to vectors for bag-of-words, sequences of tokens for recurrent neural networks, and directly supplying them as graphs to graph neural networks (Section 3).

We compare the effectiveness of these four families of ML approaches with different combinations of data preparation routines. 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 by 4%-7%. Second, with more precise data preparation, we achieved large performance improvements over the state-of-the-art (Koc et al., 2017) with our highest accuracy for each dataset reaching at least 80% all the way up to 100% for the OWASP dataset. 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 5).

This work extends our previous work in ICST 2019 (Koc et al., 2019). In that work, we collected two datasets of bug reports produced by FindSecBugs (Arteau et al., 2018), comprising 2371 reports in a synthetic benchmark and 400 manually classified reports from real-world programs. Given these reports, we proposed and implemented various data preparation techniques in order to adapt four families of ML approaches (hand-engineered features, bag-of-words, recurrent neural networks, and graph neural networks) to the task of triaging reports. These data preparation techniques were based on different program representations, including hand-engineered features and various preparations of the PDG. We performed large-scale experiments, manually comparing the effectiveness of these four families of ML approaches with different combinations of data preparation routines. We found that neural network approaches tended to perform best across all dataasets.

This article makes the following additional contributions:

– We augmented our previous datasets with reports from two new tools, which verify C and Java programs. To evaluate the effectivenss of ML techniques on these tools, we augmented our benchmark dataset with 1368 new benchmark cases.

– We present new data preparation techniques based on new program representations that were not applicable in the previous work. Our data preparation techniques are now applicable to both program slices, represented as subgraphs of the PDG, and to ASTs in the case that program slicing is infeasible (e.g., there is no line number in the bug report).
– We performed large-scale experiments on our augmented datasets to empirically compare the effectiveness of the different ML approaches. We provide more generality to our previous results, as we still found that neural networks continue to perform best across both languages and all three tools, and that different data preparation routines can improve classification accuracy by up to 7.6%.

2 Tools and Benchmarks

We studied three SA tools—FindSecBugs (Arteau et al., 2018), CBMC (Kroening and Tautschnig, 2014), and JBMC (Cordeiro et al., 2018)—which verify correctness or detect bugs in programs across two languages, C and Java. FindSecBugs is a security checker for Java web applications that can check for over 100 different bug patterns (Arteau et al., 2018). CBMC and JBMC are model checkers for C and Java, respectively, that can check various properties of a program, such as whether it terminates or whether it contains buffer overflows (Cordeiro et al., 2018; Kroening and Tautschnig, 2014). We selected these tools because they represent different program analysis approaches across different languages, allowing us to draw more general conclusions from our evaluation. In the remainder of this section, we focus on the specific configuration of each tool, and specify the ground-truth datasets we used for the evaluation.

2.1 FindSecBugs

The first SA tool we study is FindSecBugs (Arteau et al., 2018) (version 1.4.6). We configured FindSecBugs to detect SQL, command, CRLF, and LDAP injections; cross-site scripting (XSS); and path traversal (XPATH) vulnerabilities. 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 (OWASP, 2014), which has been used to evaluate various SA tools in the literature (Burato et al., 2017; Koc et al., 2017; Xypolytos et al., 2017). In particular, we used the same programs as Koc et al. (2017) 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.

The second benchmark we used consists of 14 real-world programs. Since this is a benchmark we constructed in our original publication, we call this the ICST benchmark. 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:

Table 1: Programs in the ICST benchmark.

Program	Description	LoC	# reports TP	# reports FP
Apollo-0.9.1	distributed config. (Apollo, 2018)	915 602	4	6
BioJava-4.2.8	comp. genomics frmwrk(Prlić et al., 2012)	184 040	26	32
FreeCS-1.2	chat server (Andres, 2013)	27 252	10	0
Giraph-1.1.0	graph processing sys. (Giraph, 2020)	120 017	1	8
H2-DB-1.4.196	database engine (h2db, 2022)	235 522	17	30
HSQLDB-2.4.0	database engine (The HSQL Development Group, 2021)	366 902	43	15
Jackrabbit-2.15.7	content repository (The Apache Software Foundation, 2022)	416 961	1	6
Jetty-9.4.8	web server w/servlets (Eclipse Foundation, 2022)	650 663	12	4
Joda-Time-2.9.9	date and time frmwrk (Joda.org, 2021)	277 230	2	3
JPF-8.0	symbolic execution tool (NASA Ames Research Center, 2022)	119 186	15	27
MyBatis-3.4.5	persistence frmwrk (MyBatis, 2021)	133 600	3	15
OkHttp-3.10.0	Android HTTP client (Block, Inc., 2022)	60 774	10	2
UPM-1.14	password management (Smith, 2019)	6358	2	13
Susi.AI-07260c1	artificial intel. API (Susi.ai, 2018)	65 388	47	46
Total		-	194	206

- 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 (Mohr et al., 2021).

Table 1 shows the details of the collected programs. This table shows information that is up-to-date as of October 2018, which is when we collected them. Several programs have been used in past research: H2-DB and Hsqldb are from the DaCapo Benchmark (Blackburn et al., 2006) (the other DaCapo programs did not satisfy our selection criteria) and FreeCS and UPM were used by Johnson et al. (2015). These 14 programs range from 6K to 916K LoC and cover a wide range of functionalities (see the description column). Two programs, Freecs and HSQDB, were downloaded from `sourceforge.net` and, as of October 2018, have 41K+ and 1M+ downloads, respectively. The remaining 12 programs were downloaded from GitHub and have a large user base with 5363 watchers, 24 723 stars, and 10 561 forks (as of Oct 2018).

Running FindSecBugs on these programs resulted in more than 400 vulnerability reports. We labeled the reports by manually reviewing the code, resulting in 194 true and 206 false positives as ground truths. 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 sanitization or safety check—indicating a true positive—or we exhaust the call tree without identifying any tainted or unchecked data-flow—indicating a false positive. One author performed most of the labeling work, while other authors verified a random subset of classifications.

Through this review process, we observed that the false positives we found in the ICST 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 ICST benchmark, we mostly see only the second and third scenarios. This demonstrates the importance of creating a real-world benchmark for our study.

2.2 CBMC/JBMC

CBMC and JBMC can check various kinds of program properties, such as whether a program terminates or whether it contains buffer overflows. Thus, these SA tools report that programs are either SAFE or UNSAFE with respect to the checked properties. As benchmarks for these verifiers, we took programs from the annual competition SV-COMP (Beyer, 2018, 2019), in which developers can submit C and Java verifiers, and to which CBMC and JBMC are regularly submitted. The SV-COMP corpus is, to our knowledge, the largest single collection of verification tasks for C and Java for which the ground truths are known. The SV-COMP dataset is an aggregation of multiple independent artificial benchmarks from various contributors. We used all 368 Java benchmarks included in the corpus, of which 204 (55.4%) are unsafe, and sampled 1000 C programs, of which 517 (51.7%) are unsafe. The sample we used was the same as the one generated by Koc et al., for their work on SATune (Koc et al., 2021). Since the ground truths are already known from the benchmark, we did not perform manual classification as was necessary for the FindSecBugs' ICST benchmark.

Both CBMC and JBMC come with various command-line parameters to tune their behavior. As observed by Koc et al., these configuration options significantly change the behavior of the verifiers (Koc et al., 2021). For these experiments, we aimed to select a configuration that would produce a balanced dataset, i.e., classify approximately the same number of programs correctly (i.e., as unsafe if a fault is present, or safe if a fault is not present) and incorrectly. We went through the configurations as Koc et al. (2021) generated from their three-way covering array in order to find such configurations. For CBMC, we used the ground truths obtained from running with the configuration *--no-assumptions --no-built-in-assertions --no-self-loops-to-assumptions --refine --slice-formula --depth 100 --unwind 100 --min-null-tree-depth 20 --paths fifo --mm tso --reachability-slice-fb --round-to-minus-inf --mathsat*, which produced 20 inconclusive results (i.e., the tool timed out in 1 minute without producing a result), 457 incorrect results, and 523 correct results. We discarded the 20 programs upon which CBMC produced an inconclusive result, effectively making our CBMC dataset 980 C programs. For JBMC, we found 24 configurations that produced the same distribution of 204 incorrect results and 164 correct results. The configuration we finally used for JBMC was *–disable-uncaught-exception-check –drop-unused-functions –full-slice –java-threading –java-unwind-enum-static –*

no-assumptions –no-pretty-names –no-self-loops-to-assumptions –nondet-static –slice-formula –string-non-empty –depth 100 –unwind 100 –max-nondet-array-length 100 –max-nonet-string-length 100 –max-nondet-tree-depth 100 –java-max-vla-length 10 –paths lifo –arrays-uf-never –reachability-slice –yices.

Note that since the SV-COMP corpus includes both safe and unsafe programs, CBMC and JBMC could exhibit true or false negatives in addition to true or false positives. Here a *true negative* means judging a safe program as SAFE, and a *false negative* means judging an unsafe program as SAFE. For this reason, the ML models for CBMC and JBMC must differentiate between true results (i.e., true positives or true negatives) and false results (i.e., false positives or false negatives) whereas for FindSecBugs, the models must only differentiate true and false positives, since the benchmarks for FindSecBugs consist solely of labeled bug reports. We hereafter refer to the datasets for CBMC and JBMC as the CBMC and JBMC datasets, respectively.

3 Adapting ML to Classify Incorrect Results

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 these approaches' use in the software engineering literature. Then we describe how we apply them to the SA incorrect result classification task with data preparation routines we developed or adopted.

HEF can be regarded as the state-of-the-practice ML approach for this problem (Wang et al., 2018). However, by design, this approach 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 for SA incorrect result classification before, while RNN has been used in a recent case study (Koc et al., 2017). Furthermore, previous work has not focused on the details of the data preparation routines or their effects on training performance and generalizability. For each approach, we describe the adapatations we made to handle the flow-based logs emitted by FindSecBugs and the bug reports emitted by CBMC and JBMC.

Detecting incorrect results can be framed as a standard binary classification problem (Russell and Norvig, 2016). 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 an incorrect result (false positive or false negative), and $y = 0$ otherwise. Constructing such a classifier requires defining an input vector $\vec{x}$ that captures features of programs that might help detect incorrect results. 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 incorrect and correct results $\{(x_1, y_1) \ldots (x_N, y_N)\}$.

3.1 Learning with Hand-engineered Features

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 a correct or incorrect result. Each property can then be represented numerically by one or more elements in $\vec{x}$. Indeed, researchers have used this approach to classify SA false positives in prior work (Heckman, 2007, 2009; Tripp et al., 2014; Yüksel and Sözer, 2013).

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.

FindSecBugs. We adapted the original feature set from Tripp et al. (2014), who 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*, and (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. We dropped the features *source URL*, *sink URL* and *external objects* as they do not appear in Java applications (our datasets consist of Java programs, see Section 2). We added 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.

CBMC/JBMC. We adopted Koc et al.'s approach for feature extraction, which they performed on the same CBMC/JBMC dataset (Koc et al., 2021). For C and Java benchmarks, we converted the source code to LLVM (LLVM Team, 2020) and WALA (IBM, 2006) intermediate representations (IRs), respectively, and counted the occurrence of each type of instruction (46 LLVM instructions and 23 WALA IR instructions). Furthermore, we counted the occurrences of different program constructs, such as loops and method calls. Specifically, for JBMC, we collected 9 constructs: (1) *number of ifs*, (2) *number of variables*, (3) *number of functions defined*, (4) *number of calls*, (5) *number of loops*, (6) *number of variables with an array type*, (7) *number of variables with a composite type*, (8) *number of variables with a fundamental type*, and (9) *number of lines*. For CBMC, we counted 13 contstructs: (1) *number of variables*, (2) *number of ifs*, (3) *number of loops*, (4) *number of function definitions*, (5) *number of function calls*, (6) *number variables with a fundamental type*, (7) *number of variables with an array type*, (8) *number of variables with a pointer type*, (9) *number of variables with a composite type*, (10) *number of composite features*,[1]

[1] Number of composite features include counting the number of variables, ifs, loops, functions defined, functions called, loads, and stores.

```c
extern void __VERIFIER_error() __attribute__ ((__noreturn__));
void __VERIFIER_assert(int cond) { if(!(cond)) { ERROR: __VERIFIER_error(); } }
extern int __VERIFIER_nondet_int();
#define size 10000
int main()
{ int a[size];
  int b[size];
  int i = 0;
  int j = 0;
  while( i < size )
  { b[i] = __VERIFIER_nondet_int();
    i = i+1; }
  i = 0;
  while( i < size )
  {     a[j] = b[i];
        i = i+1;
        j = j+1; }
  i = 0;
  j = 0;
  while( i < size )
  {     __VERIFIER_assert( a[j] == b[j] );
        i = i+1;
        j = j+1;}
  return 0; }
```

(a) A C program from CBMC dataset.

```
filename: array-examples/standard_copy1_true-unreach-call_ground.i
numVars: 7, numIfs: 4, numLoops: 3, numFuncs: 2, numCalls: 3, numFundTypes: 5,
     numArrayTypes: 2, numPointerTypes: 0, numCompTypes: 0,
     numCompositeFeatures: 41, numLines: 71, numLoads: 15, numStores: 10,
     numAddrSpaceCast: 0, numAlloca: 0, numAtomicCmpXchg: 0, numAtomicRMW: 0,
     numBitCast: 0, numBranch: 14, numCall: 3, numCatchPad: 0, numCatchReturn:
     0, numCatchSwitch: 0, numCleanupPad: 0, numCleanupReturn: 0,
     numExtractElement: 0, numExtractValue: 0, numFCmp: 0, numFence: 0,
     numFPExt: 0, numFPToSI: 0, numFPToUI: 0, numFPTrunc: 0, numGetElementPtr:
     5, numICmp: 5, numIndirectBr: 0, numInsertElement: 0, numInsertValue: 0,
     numIntToPtr: 0, numInvoke: 0, numLandingPad: 0, numLoad: 15, numPHINode:
     0, numPtrToInt: 0, numResume: 0, numReturn: 2, numSelect: 0, numSExt: 5,
     numShuffleVector: 0, numSIToFP: 0, numStore: 10, numSwitch: 0, numTrunc:
     0, numUIToFP: 0, numUnreachable: 1, numVAArg: 0, numZExt: 1,
Label: TrueResult
```

(b) The corresponding HEF features.

Fig. 1: An example of the HEF generated for a C program in the CBMC dataset.

(11) *number of lines*, (12) *number of load instructions*, and (13) *number of store instructions*. We present examples of how the HEF features are calculated from C and Java programs in Figures 1 and 2 respectively. Next, we explore how to represent program source code for more complex ML approaches that can implicitly learn feature representations.

```java
import org.sosy_lab.sv_benchmarks.Verifier;

public class Main {
  public static void main(String[] arg) {
    int i = 0;
    boolean b = Verifier.nondetBoolean();

    while (true) {
      i++;
      assert (b);
    }
  }
}
```

(a) A Java program from JBMC dataset.

```
entryClass: jayhorn-recursive/InfiniteLoop
numIfs: 2, numVars: 10, numFuncs: 2, numCalls: 2, numLoops: 2, numArrayTypes:
     0, numCompositeTypes: 1, numFundamentalTypes: 9, numLines: 14,
     SSASwitchInstruction: 1, SSAArrayLengthInstruction: 0,
     SSAArrayLoadInstruction: 0, SSAPutInstruction: 0,
     SSAComparisonInstruction: 1, SSAConditionalBranchInstruction: 1,
     SSANewInstruction: 0, SSAReturnInstruction: 0, SSAGetInstruction: 0,
     SSAGotoInstruction: 3, SSALoadMetadataInstruction: 1,
     SSAInstanceofInstruction: 1, SSAArrayStoreInstruction: 0,
     SSACheckCastInstruction: 1, SAGetCaughtExceptionInstruction: 0,
     SSABinaryOpInstruction: 3, SSAPhiInstruction: 0, SSAUnaryOpInstruction:
     1, SSAConversionInstruction: 0, SSAPiInstruction: 0,
     SSAMonitorInstruction: 0, SSAInvokeInstruction: 1, SSAThrowInstruction: 2,
Label: FalseResult
```

(b) The corresponding HEF features.

Fig. 2: An example of the HEF generated for a Java program in the JBMC dataset .

3.2 Representing Programs

For HEF approaches, we extract feature vectors directly from the source code. However, for BOW, LSTM, and GGNN approaches, a program representation that is suitable for learning is required. We use two different program representations, one for the programs analyzed by FindSecBugs, and one for the programs analyzed by CBMC and JBMC.

Each report generated by FindSecBugs contains a source line, indicating the location of the bug. We can utilize this information to narrow down the code to the most relevant parts, removing noise from the input data. We use Koc et al.'s preprocessing step (Koc et al., 2017), which is to compute a backward slice (Weiser, 1981) of the programs being analyzed starting from the source line in the SA report. The backward slice includes all statements that may affect the behavior at the reported line, hence it should contain highly relevant information for SA report classification. This is especially useful for the ICST benchmark, which is composed of large, real-world programs (see Table 1).

This approach works for FindSecBugs reports, but not for CBMC and JBMC reports, as the latter do not always include a source line. For example, termination

```
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. 3: Sample PDG node (simplified for presentation).

checks do not emit a faulty line number if the program is determined to violate that property. Thus, without a consistent slicing criterion for these tools' reports, we instead use the full source code as input, represented as an AST.

FindSecBugs. We computed backwards slices for FindSecBugs results using Joana (Mohr et al., 2021), a program analysis framework for Java. The first step is determining the entry point(s) from which the program starts to run. For our problem, we first generate the call hierarchy of the method containing the error line. We then identify in this hierarchy the methods that can be invoked by the user, and set those as the entry points. Such methods can be APIs if the program is a library, 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).

Figure 3 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. Below, we will refer to these fields of PDG nodes.

CBMC/JBMC. We generated the ASTs for the programs analyzed by CBMC and JBMC. For C programs, we used Clang version 12.0.0 (The Clang Team, 2021) to generate the AST. For Java programs, we used ASTExtractor version 0.5 (Diamantopoulos, 2020). Figure 4 shows an example of an AST we use. In line 1, we see the target variable which denotes the program label. At line 2, we see the entire the AST graph structure represented using edges, `[0,1]` represents an edge from `node 0` to `node 1`. At line 3, we see the Node Type for all the nodes present in the AST. For example, `node 0` is of the type `CompilationUnit`. At line 4, we see the node content for all the nodes in the AST, which represents the actual content from the code within each node. For `node 0` there is no corresponding node content but for `node 1` which is of the type `ImportDeclaration`, the node content is `import org.sosy_lab.sv_benchmarks.Verifier;`

[2] Joana uses the intermediate representation from the T.J. Watson Libraries for Analysis (*WALA*) (IBM, 2006).

```java
import org.sosy_lab.sv_benchmarks.Verifier;

public class Main {
  public static void main(String[] arg) {
    int i = 0;
    boolean b = Verifier.nondetBoolean();

    while (true) {
      i++;
      assert (b);
    }
  }
}
```

(a) A Java program from the JBMC dataset.

```
targets: [[0, 1]],
graph: [[0,1], [0,2], [2,3], [3,4], [4,5], [4,6], [3,7], [3,8], [8,9], [9,10],
     [9,11], [9,12], [9,13], [8,14], [14,15], [14,16], [16,17], [17,18],
     [18,19], [17,20], [20,21], [3,22], [3,23], [2,24], [2,25]],
Node_Type: ["CompilationUnit", "ImportDeclaration", "TypeDeclaration",
     "MethodDeclaration", "SingleVariableDeclaration", "ArrayType",
     "SimpleName", "SimpleName", "Block", "VariableDeclarationStatement",
     "VariableDeclarationFragment", "PrimitiveType",
     "VariableDeclarationFragment", "PrimitiveType", "WhileStatement",
     "BooleanLiteral", "Block", "ExpressionStatement", "PostfixExpression",
     "SimpleName", "MethodInvocation", "SimpleName", "PrimitiveType",
     "Modifier", "SimpleName", "Modifier"],
Node_Content: ["", "import org.sosy_lab.sv_benchmarks.Verifier;", "", "", "",
     "String[]", "arg", "main", "", "", "i=0", "int",
     "b=Verifier.nondetBoolean()", "boolean", "", "True", "", "", "", "i", "",
     "assert b", "void", "public static", "Main", "public"]
```

(b) The entire corresponding AST.

Fig. 4: An example of the AST generated for a Java program in the JBMC dataset .

3.3 Bag of Words

One of the approaches to learn directly from the program data is to transform it into a feature vector such that it would useful summarized information to detect incorrect results. For this, 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 (Goldberg, 2017). 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 the software engineering literature as an information retrieval technique to solve problems such as duplicate report detection (Sureka and Jalote, 2010), bug localization (Lukins et al., 2010), and code search (Ye et al., 2016). 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.

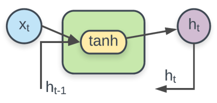
FindSecBugs/CBMC/JBMC. In our experiments, we used two variations of BoW. The first variation checks the occurrence of words, which leads to a binary feature vector representation, where the features are the words. 1 means that the corresponding word is in a program, and 0 means it is not. The second 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 (for FindSecBugs) or ASTs (for CBMC/JBMC) using data preparation routines introduced in Section 3.4 for SA reports.

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.

3.4 Recurrent Neural Networks

BoW features ignore order. For text classification, recurrent neural networks (RNNs) (GERS et al., 2000; Hochreiter and Schmidhuber, 1997; Mandic and Chambers, 2001) have emerged as a powerful alternative approach that views text as an (arbitrary-length) sequence of words and automatically learns vector representations for each word in the sequence (Goldberg, 2017).

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 vector $\langle h_0, h_1, \ldots, h_n \rangle$ 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 (Dam et al., 2016) and code synthesis (Kushman and Barzilay, 2013; Ling et al., 2016). In a recent paper, Koc et al. (2017) conducted a case study using Long Short-term Memories (LSTM) (GERS et al., 2000; Hochreiter and Schmidhuber, 1997), a popular kind of RNN, for classifying SA false positives.

FindSecBugs/CBMC/JBMC. In this work, we study LSTMs as well, as they are well suited to modeling long sequences (Sak et al., 2014). To use LSTM, we need to transform program representations 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. For each transformation we denote the results it was applied to in parentheses.

– *Data Cleansing (T_{cln})* (FindSecBugs). 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 the program's source code. These nodes are removed because they do not provide useful information for learning. Some of them do not even exhibit 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`). We don't perform this operation for CBMC and JBMC as we create ASTs for these tools and ASTs contain no redundant information.
– *Data Tokenization (T_{tkn})* (FindSecBugs/CBMC/JBMC). After performing the above operation for FindSecBugs, we extract tokens from paths of classes and methods by splitting them by '.' or '/'. For CBMC we use Clang version 12.0.0 to tokenize the programs, while for JBMC we use the javalang Python library (Thunes, 2020).
– *Abstracting Numbers and String Literals (T_{ans})* (FindSecBugs/CBMC/JBMC). These transformations replace numbers and string literals that appear in a program slice or AST 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 with *N3*, and numbers with four or more digit 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 with the token `STR` followed by a unique number. For example, the first string literal in the list will be replaced with `STR1`.
– *Abstracting Program-specific Words (T_{aps})* (FindSecBugs/CBMC/JBMC). 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. Therefore, T_{aps} abstracts away certain words from the dataset that occur less frequently, or that only occur in a single program, by replacing them with phrase *UNK*. Similar to T_{ans}, these transformations may improve the effectiveness by reducing the vocabulary size and generalizability via abstractions.
– *Extracting English Words From Identifiers (T_{ext})* (FindSecBugs/CBMC/JBMC) Many identifiers are composed of multiple English words. For example, `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

[3] A property of the representation which requires that each variable is assigned exactly once, and every variable is defined before it is used (Rosen et al., 1988).

size, we split any `camelCase` or `snake_case` identifiers into their constituent words.

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. (2017). 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.

3.5 Graph Neural Networks

With RNNs, 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 (Gori et al., 2005; Scarselli et al., 2009). 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 n_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 $\mathrm{NBR}(n_i)$ at each time step t, i.e, $h_i^{(t)} = f(n_i, h_{\mathrm{NBR}(n_i)}^{(t-1)})$. The function f is defined as a neural network.

In our study, we focus on a variation of GNNs called Gated Graph Neural Networks (GGNN) (Li et al., 2015). GGNNs have gated recurrent units and enable initialization of the node representation. GGNNs have been used to learn properties about programs (Allamanis et al., 2017; Li et al., 2015), but have rarely been applied to classify SA false positives or to learn from program slices or ASTs. To adapt GGNNs to our problem, we explore several different node representations:

FindSecBugs.

- *Using Kind, Operation, and Type Fields (KOT).* As the first representation, we only use the `Kind`, `Operation`, and `Type` fields of the graph nodes. For the example node in Figure 3, the KOT node representation is V_{rep} =[`EXPR`, `reference`, `Ljava/sql/Connection`]
- *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 3 is V_{rep} =[`EXPR`, `reference`, `Ljava/sql/Connection`, `object`] (`object` is the extracted item, meaning that the reference is for an object).
- *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 an 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 (Goldberg and Levy, 2014; Mikolov et al., 2013). 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.

CBMC and JBMC.

– *Using Type Field (T).* As the first representation, we only use the `Type` fields of the graph nodes. For the example AST in Figure 4b, for the first node, the T node representation is V_{rep} = [`Encoder(Node_Type[0])`] i.e. [`Encoder(CompilationUnit`], where Encoder is used to Encode Node Type features using an ordinal encoding scheme.
– *Extracting First N Tokens in Addition to T (NT).* In the second representation, in addition to the Type field, we tokenize and encode the tokens derived from the `NodeContent` field of the AST for a particular node. The `N` attribute in the name is used to represent the number of tokens being considered for each node. The operation can be represented as $V_{rep} = E_T + \sum_{i=0}^{N} Encode(x_i)$, where +, Σ are the concatenation operations and x_i represent tokens for a particular node's content. To create tokens from the `NodeContent` field, we perform the same operations as aforementioned in the LSTM section. If a particular node has k tokens, where $k < N$, we concatenate $[-1] * N - k$ times to V_{rep}, thus ensuring a constant size of N+1 for node representation.
– *Type Field and Node Encoding Using Embeddings (ET).* In the third representation, we follow mostly the same procedure as that for FindSecBugs. For each node, we compute a vector representation (E_V) using word embeddings from a *word2vec* model (Goldberg and Levy, 2014; Mikolov et al., 2013) for all the tokens present in that node's content `NodeContent`). The operation can be represented as $E_V = \frac{1}{N} \sum_{i=0}^{N} Word2Vec(x_i)$, where Σ is the addition operation and x_i represent tokens for a particular node's content. We create the final node vector V_{rep} by concatenating E_V with the embedding vectors of `type`, i.e., $V_{rep} = E_T + E_V$, where + is the concatenation operation.

4 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 3. For learning with HEF, we experimented with 9 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 3.1. We used the WEKA (Eibe et al., 2016) implementations of these algorithms.

Table 2: Preparations, their names, and the datasets to which they were applied.

Applied preparations	Approach name	Datasets
Occurrence feature vector	*BoW-Occ*	OWASP, ICST CBMC, JBMC
Frequency feature vector	*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 vector	*GGNN-KOT*	OWASP, ICST
KOT + an extracted item	*GGNN-KOTI*	
Node Encoding	*GGNN-Enc*	
Node Type	*GGNN-T*	CBMC, JBMC
Node type + first N node content tokens	*GGNN-N T*	
Node type + node content word encoding	*GGNN-EncT*	

For the other three families of approaches, we experimented with the variations described in Sections 3.3, 3.4, and 3.5. Table 2 lists these variations with their names and the 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. For BoW and HEF, we also used the AutoML package (AutoML, 2022) for hyper-parameter tuning. We adapted the LSTM implementation designed by Carrier et al. (2018) and extended the GGNN implementation from Microsoft Research (Microsoft, 2019).

As shown in Table 2, for OWASP and ICST datasets, we used the GGNN-KOT, GGNN-KOTI, and GGNN-Enc algorithms. For the CBMC and JBMC datasets we used the GGNN-T, GGNN-NT and GGNN-EncT algorithms. LSTM, HEF and BoW approaches were common for all the datasets.

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 ICST benchmark randomly into training and test sets. Thus, both training and test sets will have samples from each program in the dataset. We refer to the ICST random split dataset as ICST-Rand for short.

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 the 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 ICST program-wise split dataset as ICST-PW for short. The CBMC and JBMC benchmarks fit into the second scenario, since programs in the benchmark are split such that the training and the test set of programs are mutually exclusive.

Table 3: The training configuration for each dataset.

Datasets	# splits	# seeds	# algorithms	# models
OWASP, ICST-Rand, ICST-PW	5	5	18	1350
CBMC, JBMC	5	5	20	1000
Total unique values			23	2350

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. Table 3 shows the training configuration for our datasets. 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. We use 4 subsets for training and 1 subset for testing, then rotate such that each subset is used as the test set once. Furthermore, we repeat each execution 5 times with different random seeds. The purpose of these many repetitions is to evaluate whether the results are consistent (see Section 5.3).

LSTM and GGNN are trained using an iterative algorithm that requires users to provide a stopping criterion. We set a timeout of 5 hours and we ended training before the timeout if there was no accuracy improvement for 20 and 100 epochs, respectively, for LSTM and GGNN (these values are referred to as the *epoch patience*). For the CBMC and JBMC datasets, we set the epoch patience to 50 and 300 for GGNN and LSTM, respectively, due to their tendency to converge more slowly. 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 preliminary experiments of 15 epochs with the ICST-Rand dataset and *LSTM-Ext* to determine the word embedding dimension size for tokens and batch size. We tested $4, 8, 12, 16, 20$, and 50 for the word embedding dimension and $1, 2, 4, 8, 16$, and 32 for the batch size. We observed that word embedding dimension 16 and batch size 12 led to the highest test accuracy on average and thus we use these values in the remaining experiments. We also used the embedding dimension of 12 for the pre-training of *GGNN-Enc*. We use dimension size 1 for encoding Tokens in *GGNN-NT*. We choose the value for N variable based on the minimum and maximum number of tokens per node in the datasets, we therefore choose $5, 8$, and 16 for CBMC and $1, 2$, and 5 for JBMC.

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 into memory, the time to test a data point is negligible (around a second) for all ML algorithms. To evaluate effectiveness, we use *precision*, *recall*, and *accuracy* as follows:

$$Precision(P) = \frac{\text{\# of correctly classified true positives}}{\text{\# of samples classified as true positive}}$$
$$Recall(R) = \frac{\text{\# of correctly classified true positives}}{\text{\# of true positives in dataset}}$$
$$Accuracy(A) = \frac{\text{\# of correctly classified samples}}{\text{\# of all samples, i.e., size of test set}}$$

Accuracy is a good indicator of effectiveness for our study because there is no trivial way to achieve high accuracy if there is 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.

- **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?

Experiments on the benchmarks corresponding to FindSecBugs results were run 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. Experiments on the CBMC and JBMC data were conducted on a server with 376GB of RAM and 2 Intel Xeon Gold 5218 16-core CPUs @ 2.30GHz running Ubuntu 18.04.

5 Analysis of Results

As seen in Table 3, we trained 2350 SA report classification models in total. The summary of the results can be found in Tables 4 and 5, 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.

5.1 RQ1: Overall Performance Comparison

In this section, we analyze the overall performance of four main learning approaches using the accuracy metric. We note that the trends we discuss here also hold for the recall and precision metrics.

In Table 4, we separate high performing approaches from others with a dashed line at points where there is a large gap in accuracy. Overall, *LSTM and GGNN 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 5.4). The GGNN variations have a large difference in accuracy. *GGNN-Enc* achieves 94%, while the

Table 4: Recall, precision and accuracy results for the approaches in Table 2 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	
	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
OWASP	*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
	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
ICST-Rand	*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
	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
ICST-PW	*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
	GGNN-NT-16	86.60	0.85	80.20	1.25	83.40	0.75
	LSTM-ANS	83.00	1.02	79.74	0.42	81.37	0.74
	LSTM-APS	83.19	1.16	79.57	0.44	81.36	0.90
	LSTM-Ext	82.85	0.87	79.55	0.50	81.25	0.42
	LSTM-Raw	82.60	1.27	79.10	1.21	81.13	1.42
	GGNN-NT-8	83.60	1.55	77.60	1.67	80.60	1.25
	HEF-RandomForest	79.50	1.35	80.60	1.65	79.50	1.35
CBMC	*HEF-K**	79.50	1.10	79.90	1.05	79.50	1.1
	BoW-Occ	79.20	1.15	79.20	1.65	79.20	1.15
	BoW-Freq	80.40	0.25	78.80	0.95	79.20	0.95
	HEF-MLP	78.60	1.65	80.50	1.90	78.60	1.65
	HEF-J48	78.30	0.30	79.70	0.70	78.36	0.3
	GGNN-NT-5	79.40	1.20	78.00	0.65	78.20	0.65
	GGNN-EncT	76.40	0.75	76.00	0.50	76.20	0.25
	GGNN-T	76.20	1.20	75.40	0.75	75.80	0.50
	GGNN-NT-5	82.10	3.05	77.60	3.50	80.15	3.30
	GGNN-NT-2	81.50	3.00	77.50	3.50	79.50	3.23
	LSTM-APS	78.60	5.24	77.97	2.23	78.50	2.39
	GGNN-NT-1	81.00	1.55	75.00	7.50	78.00	2.25
	LSTM-Ext	78.68	6.25	76.45	2.41	77.72	1.71
	LSTM-ANS	78.44	5.33	76.45	3.20	77.44	3.00
	LSTM-Raw	78.68	11.22	76.98	2.36	77.44	3.06
JBMC	*BoW-Freq*	73.70	1.35	75.70	1.40	74.70	1.35
	BoW-Occ	74.30	2.20	74.40	2.45	74.30	2.2
	HEF-J48	74.00	2.35	74.00	2.45	74.00	2.35
	HEF-RandomForest	73	4.4	73.80	4.15	73	4.4
	GGNN-EncT	72.20	2.55	75.90	3.50	73.95	3.30
	GGNN-T	72.00	2.45	75.90	3.50	73.75	3.20
	HEF-RandomForest	73.00	4.4	73.80	4.15	73.00	4.4
	HEF-MLP	64.90	1.85	66.70	2.95	64.90	1.85

Table 5: Number of epochs and training times for the LSTM and GGNN approaches. Median and SIQR values are shown as in Table 4.

Dataset	Model	# 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
ICST-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
ICST-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
CBMC	*LSTM-Raw*	115 5	8045 857
	LSTM-ANS	100 8	7491 414
	LSTM-APS	110 4	7273 256
	LSTM-Ext	95 6	7333 297
	GGNN-NT	75 12	974 73
	GGNN-EncT	62 4	834 21
	GGNN-T	64 6	856 45
JBMC	*LSTM-Raw*	91 18	233 71
	LSTM-ANS	93 14	181 12
	LSTM-APS	97 11	171 23
	LSTM-Ext	94 16	287 31
	GGNN-NT	545 221	25 6
	GGNN-EncT	201 45	12 3
	GGNN-T	212 76	14 2

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 ICST-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 ICST-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 6 shows the dictionary sizes for the LSTM approaches, and Table 7 shows the length of samples for the LSTM approaches (the 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 on the OWASP dataset, the number of nodes is

24 on average and 82 at most, the number of edges is 47 on average and 174 at most. The ICST dataset is significantly larger both in dictionary size and sample lengths, resulting in 1880 average to 16 479 maximum nodes, and 6411 average to 146 444 maximum edges.

For the ICST-PW dataset, all accuracy results except LSTM-Ext are below 80%. Recall that this split was created for the second application scenario where 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, we observe that both HEF and BoW are very efficient. All their variations complete training in less than a minute for all datasets, while the LSTM and GGNN approaches run for hours for the ICST-Rand and ICST-PW datasets (Table 5). This is mainly due to the large number of parameters being optimized in LSTM and GGNN. However for most of the benchmarks and scenarios, these neural network approaches tend to achieve the highest accuracy.

For the CBMC and JBMC datasets, the LSTM and GGNN approaches typically achieve high accuracy compared to the other approaches. The GGNN-NT approach achieves the highest accuracy in both cases with N = 16 and N = 5 for CBMC and JBMC, respectively. However, certain GGNN approaches also achieve relatively low accuracy for both benchmarks compared to other ML approaches. For example, GGNN-T achieved the lowest accuracy and the third-lowest accuracy in CBMC and JBMC, respectively. This emphasizes the importance of data preparation for GGNN. The LSTM approaches tend to perform better than HEF and BoW, suggesting that neither the hand-engineered features nor the textual content of the programs are adequate for these application scenarios. This is consistent with what we have seen with results for the previous datasets.

Lastly, note that the results on the OWASP dataset (Table 4) are directly comparable with the results reported by Koc et al. (2017), 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.

Achieving high performance with the OWASP benchmarks is good to demonstrate that some of ML-based classification approaches is capable of learning relevant API usage and complex coding patterns. However, the performance on such a benchmark by itself is not sufficient to demonstrate generalizability alone. For this reason, we have included multiple benchmarks and tools in our empirical assessment study.

5.2 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

Table 6: Dictionary sizes for the LSTM approaches.

	Dictionary size			
Approach	OWASP	ICST	CBMC	JBMC
LSTM-Raw	333	13 237	13 503	942
LSTM-ANS	284	9724	8420	842
LSTM-APS	284	9666	4053	384
LSTM-Ext	251	4730	3404	381

Table 7: Sample lengths for the LSTM approaches. Numbers in the normal font are the maximum and in the smaller font are the mean.

	Sample length							
Approach	OWASP		ICST		CBMC		JBMC	
LSTM-Raw	735	224	156 393	18524	90 923	10134	2425	533
LSTM-ANS	706	212	149 886	18104	90 923	10133	2425	532
LSTM-APS	706	212	150 755	18378	90 923	10133	2425	532
LSTM-Ext	925	277	190 950	23031	91 009	10637	2473	539

information that is available in the program context. *We found LSTM-Ext produced the overall best accuracy results across for the OWASP, ICST-Rand, and ICST-PW datasets. The different node representations of GGNN present tradeoffs, while the BoW variations produced similar results. GGNN-NT produced the best overall best accuracy results for CBMC and JBMC, with the different LSTM data preparation techniques just behind.*

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 ICST-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 ICST-PW dataset. For the CBMC and JBMC datasets, we see that data preparation routines improve the accuracy of LSTM albeit by a rather small margin (less than 1.5%). For the first application scenario it was LSTM-Raw which had the best results but for the second application scenario LSTM-APS performs better on average.

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 ICST 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 ICST-Rand, and 74%, 72%, and 74.67% accuracy on the ICST-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 1, 6 and 7). 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.

We used slightly different node representations and we also used Abstract Syntax Trees (ASTs) for representing GGNN in the CBMC and JBMC datasets. In these datasets, we observe an overall better accuracy for the GGNN-NT approach over other GGNN approaches which signifies that encoding the node content as tokens is useful information. We see an improvement to 83.80% for GGNN-NT-16 as compared to 76.20% for GGNN-EncT and 75.80% for GGNN-T when using the CBMC dataset, we similarly see a high accuracy of 80.15% for GGNN-NT-5 as compared to 73.95% and 73.00% accuracy for GGNN-EncT and GGNN-T respectively when using the JBMC dataset. Also the comparable accuracy of GGNN-EncT and GGNN-T (76.20% and 75.80% for CBMC and 73.95% and 73.00% for JBMC) denotes that word2vec encodings provide some but not much useful information for classification. Since GGNN-NT approaches also achieve the highest accuracy for the CBMC and JBMC datasets, it seems that encoding ASTs in general might be useful for learning structural information from the programs for static analysis.

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 ICST-Rand dataset. This result suggests that checking the presence of a word is almost as useful as counting its occurrences.

5.3 RQ3: Variability Analysis

In this section, we analyze the variance in the recall, precision, and accuracy results using the semi-interquartile range (SIQR) values given in the smaller font in Table 4. 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.

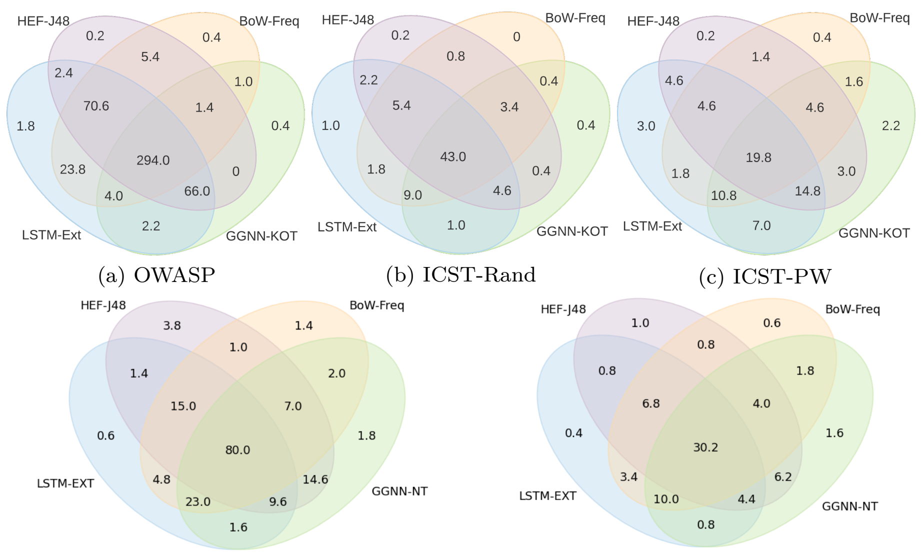
On the OWASP dataset, all approaches have little variance, except for a 7% SIQR for the recall value of HEF-MLP.

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

On the CBMC dataset, all approaches seem to have little variance.

Lastly, on the JBMC dataset, the variances are relatively high compared to CBMC, especially for some of the LSTM approaches. However, we again observe that applying more data preparation for LSTM leads to a smaller variance for all



(a) OWASP (b) ICST-Rand (c) ICST-PW

(d) CBMC (e) JBMC

Fig. 5: Venn diagrams of the number of correctly classified examples for *HEF-J48*, *BoW-Freq*, *LSTM-Ext*, and *GGNN-KOT*/*GGNN-NT* approaches, average for 5 models trained for the OWASP (a), ICST-Rand (b), ICST-PW (c), CBMC (d), and JBMC (e) datasets (474, 74, 80, 168, and 74 test samples respectively).

the metrics for the JBMC dataset. The overall best 2 approaches for JBMC, GGNN-NT-5 and GGNN-NT-2 have a SIQR of less than 4% in accuracy. The BoW and HEF approaches have an SIQR of less than 3% in accuracy, except for the HEF-RandomForest approach which seems to be an outlier.

5.4 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* (*GGNN-NT* for CBMC/JBMC) because these instances generally produce the best results in their family. Figure 5 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 5-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 ICST-Rand results in Figure 5-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 ICST-PW results in Figure 5-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.

The CBMC results in Figure 5-d show that 80 (47.9%) reports were correctly classified by all the approaches, meaning that these reports have symptoms that are detectable by all approaches. However a large portion of the reports 88 (52.1%) were misclassified by at least one approach. The JBMC results in Figure 5-e show that only 30.2 reports were correctly classified by all approaches, which is mostly due to the fact that reports often classified properly by the *LSTM-Ext* and *GGNN-NT* approaches are misclassified by either *HEF-J48* and *BoW-Freq*.

Last, the overall results in Figure 5 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 6-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 6-b shows the sequential representation used by *LSTM-Ext*. *HEF-J48* used the following feature vector:

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

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 (Figure 6-b). 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 3 corresponding to the switch statement. Then, we see string and SQL operations through lines 4 to 7, followed by a `PHI` instruction at line 8. 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 6-b without their order. This suggests that the overall correlation of the tokens that appear in this

```java
public void doPost(HttpServletRequest request, HttpServletResponse response){
  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) A program from the OWASP dataset.

```
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) Corresponding sequential representation used for *LSTM-Ext*.

Fig. 6: An example program (simplified) from the OWASP benchmark that was correctly classified only by *LSTM-Ext* and the sequential representation used for *LSTM-Ext*.

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.

5.5 Threats To Validity

There are several threats to the validity of our study. First, the benchmarks may not be representative. Indeed, the OWASP, CBMC, and JBMC benchmarks are synthetic. Therefore, we collected the first real-world (ICST) benchmark for classifying SA results, consisting of 14 programs to increase the generalizability of our results. Specifically for CBMC and JBMC, we aimed to select configurations that would produce a balanced dataset, which is necessary for training. However this could lead to a strange set of configuration options being selected that don‘t represent real usage.

In addition, our ICST 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 splits to analyze the variability that might be caused by having limited data. Second, we ran our experiments on a virtual machine and server, 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.

6 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 (Heckman, 2007, 2009; Tripp et al., 2014; Yüksel and Sözer, 2013). For instance, Tripp et al. (2014) 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. (2017) 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. (2018) 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 (White et al., 2016), API mining (Fowkes and Sutton, 2016; Gu et al., 2016), variable naming and renaming (Allamanis et al., 2015; Raychev et al., 2015), code suggestion and completion (Nguyen et al., 2013; Raychev et al., 2014; Tu et al., 2014), bug detection (Allamanis et al., 2017). Allamanis et al. (2018) conducted an extensive survey of such research efforts. There have also been a lot of work recently on using NLP techniques and deep learning to develop and update comments based on existing code and any changes which occur (Gros et al., 2020; Haque et al., 2020; Panthaplackel et al., 2020). Searching open-source repositories to retrieve existing code snippets for a given user query is a key task in software engineering, in recent years there has been an increase of interest in using natural language techniques for doing this code search, some datasets for the same problem have also been published (Li et al., 2019; Wan et al., 2019; Wang et al., 2020). There have also been research to develop unsupervised embeddings for software libraries which can then be used for any of the above mentioned tasks (Alon et al., 2019; Chen and Monperrus, 2019; Feng et al., 2020). However, none of these efforts addresses the problem of identifying and distinguishing false positive SA reports.

7 Conclusions and Future Work

We have presented an empirical study that evaluates four families of ML approaches, i.e, HEF, BoW, LSTM, and GGNN, for classifying static analysis results to classify incorrect results emitted by three SA tools: FindSecBugs, CBMC, and JBMC. To adapt these approaches to classify incorrect results, we introduced new code transformations for preparing data as inputs.

In our experiments, we compared 13 ML approaches from four families, using multiple benchmarks and 3 tools under two application scenarios. The results of our experiments suggest that neural network approaches work better than HEF and BOW approaches. The LSTM approach worked better for classifying false positives emitted by the FindSecBugs tool, while GGNN worked better for classifying incorrect results emitted by CBMC and JBMC. This may indicate that GGNNs are more suited for learning on ASTs and LSTMs are more suited for learning on program slices. We evaluated two potenital usage scenarios, and found that the application scenario in which the training set and test set contain different programs is more challenging as it requires learning the symptoms that holds across programs. However, 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. Finally, evaluated both ASTs and program slices as input to ML models, and found that both are able to effectively encode structural information, the former working better for GGNNs and the latter for LSTMs. 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. We also plan on using different embedding being developed specifically for software libraries for either node representation or as input to recurrent neural networks.

Acknowledgment

This work was partly supported by NSF grants CCF-2007314, CCF-2008905 and CCF-2047682, the NSF graduate research fellowship program, and Eugene McDermott Graduate Fellowship 202006.

References

Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/2786805.2786849

Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51, 4, Article 81 (July 2018), 37 pages. https://doi.org/10.1145/3212695

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to Represent Programs with Graphs. http://arxiv.org/abs/1711.00740. *arXiv:1711.00740 [cs]* (Nov. 2017).

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

Manfred Andres. 2013. Free Chat-server: A chatserver written in Java. https://sourceforge.net/projects/freecs.

Apollo 2018. Apollo: a distributed configuration center. https://github.com/ctripcorp/apollo.

Philippe Arteau, David Formánek, and Tomáš Polešovský. 2018. Find Security Bugs, version 1.4.6. http://find-sec-bugs.github.io, Accessed: 2018-10-02.

AutoML. 2022. AutoML. https://www.automl.org/automlhttps://www.automl.org/automl/.

Dirk Beyer. 2018. Results of the Competition. https://sv-comp.sosy-lab.org/2018/results/results-verified/, Accessed: 2021-04-22.

Dirk Beyer. 2019. Results of the Competition. https://sv-comp.sosy-lab.org/2019/results/results-verified/, Accessed: 2021-04-22.

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovi, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. 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 (OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

Block, Inc. 2022. OkHttp: An HTTP & HTTP/2 client for Android and Java applications. http://square.github.io/okhttp.

Elisa Burato, Pietro Ferrara, and Fausto Spoto. 2017. Security Analysis of the OWASP Benchmark with Julia. In *Proc. of ITASEC17, the rst Italian Conference on Security, Venice, Italy.*

Pierre-Luc Carrier and Kyunghyun Cho. 2018. LSTM Networks for Sentiment Analysis: DeepLearning 0.1 documentation. http://deeplearning.net/tutorial/lstm.html.

Zimin Chen and Martin Monperrus. 2019. A literature study of embeddings on source code. *arXiv preprint arXiv:1904.03061* (2019).

Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. 2018. JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode. In *Computer Aided Verification (CAV) (LNCS, Vol. 10981)*. Springer International Publishing, Cham, 183–190.

Hoa Khanh Dam, Truyen Tran, and Trang Thi Minh Pham. 2016. A deep language model for software code. In *FSE 2016: Proceedings of the Foundations Software Engineering International Symposium*. 1–4.

Themistoklis Diamantopoulos. 2020. ASTExtractor v0.5. https://github.com/thdiaman/ASTExtractor.

Eclipse Foundation. 2022. Jetty: lightweight highly scalable java based web server and servlet engine. https://www.eclipse.org/jetty.

Frank Eibe, MA Hall, and IH Witten. 2016. The WEKA Workbench. *Morgan Kaufmann* (2016).

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL]

Jaroslav Fowkes and Charles Sutton. 2016. Parameter-free Probabilistic API Mining Across GitHub. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 254–265. https://doi.org/10.1145/2950290.2950319

Felix A GERS, Jürgen SCHMIDHUBER, and Fred CUMMINS. 2000. Learning to forget: Continual prediction with LSTM. *Neural computation* 12, 10 (2000), 2451–2471.

Giraph 2020. Giraph : Large-scale graph processing on Hadoop. http://giraph.apache.org.

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

Yoav Goldberg and Omer Levy. 2014. word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method. *arXiv:1402.3722* (2014).

Marco Gori, Gabriele Monfardini, and Franco Scarselli. 2005. A new model for learning in graph domains. In *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, Vol. 2. IEEE, 729–734.

David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. 2020. Code to Comment "Translation": Data, Metrics, Baselining & Evaluation. arXiv:2010.01410 [cs.SE]

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 631–642.

h2db 2022. H2 Database Engine. http://www.h2database.com.

Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved Automatic Summarization of Subroutines via Attention to File Context. *Proceedings of the 17th International Conference on Mining Software Repositories* (Jun 2020). https://doi.org/10.1145/3379597.3387449

S. S. Heckman. 2007. Adaptive Probabilistic Model for Ranking Code-Based Static Analysis Alerts. In *Software Engineering - Companion, 2007. ICSE 2007 Companion. 29th International Conference on*. 89–90. https://doi.org/10.1109/ICSECOMPANION.2007.16

Sarah Smith Heckman. 2009. *A systematic model building process for predicting actionable static analysis alerts*. North Carolina State University.

Sepp Hochreiter and Jurgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (Nov. 1997), 1735.

IBM. 2006. T. J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/.

Joda.org. 2021. Joda-Time a quality replacement for the Java date and time classes. http://www.joda.org/joda-time.

Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. 2015. Exploring and Enforcing Security Guarantees via Program Dependence Graphs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 291–302. https://doi.org/10.1145/2737924.2737957

Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 672–681. http://dl.acm.org/citation.cfm?id=2486788.2486877

Ugur Koc, Austin Mordahl, Shiyi Wei, Jeffrey S. Foster, and Adam Porter. 2021. SATune: A Study-Driven Auto-Tuning Approach for Configurable Software Verification Tools. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE 2021)*. ACM.

Ugur Koc, Parsa Saadatpanah, Jeffrey S. Foster, and Adam A. Porter. 2017. 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 (MAPL 2017)*. ACM, New York, NY, USA, 35–42. `https://doi.org/10.1145/3088525.3088675`

Ugur Koc, Shiyi Wei, Jeffrey S. Foster, Marine Carpuat, and Adam A. Porter. 2019. An Empirical Assessment of Machine Learning Approaches for Triaging Reports of a Java Static Analysis Tool. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 288–299. `https://doi.org/10.1109/ICST.2019.00036`

Daniel Kroening and Michael Tautschnig. 2014. CBMC – C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 389–391.

Nate Kushman and Regina Barzilay. 2013. 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*. 826–836.

Hongyu Li, Seohyun Kim, and Satish Chandra. 2019. Neural Code Search Evaluation Dataset. arXiv:1908.09804 [cs.SE]

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated Graph Sequence Neural Networks. *arXiv:1511.05493* (Nov. 2015). `http://arxiv.org/abs/1511.05493`

Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Fumin Wang, and Andrew Senior. 2016. Latent Predictor Networks for Code Generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 599–609.

LLVM Team. 2020. The LLVM Compiler Infrastructure. `https://github.com/llvm/llvm-project.git`.

Stacy K. Lukins, Nicholas A. Kraft, and Letha H. Etzkorn. 2010. Bug localization using latent Dirichlet allocation. *Information and Software Technology* 52, 9 (2010), 972 – 990. `https://doi.org/10.1016/j.infsof.2010.04.002`

Danilo P Mandic and Jonathon Chambers. 2001. *Recurrent neural networks for prediction: learning algorithms, architectures and stability.* John Wiley & Sons, Inc.

Microsoft. 2019. Microsoft Gated Graph Neural Networks. `https://github.com/Microsoft/gated-graph-neural-network-samples`.

Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean, L Sutskever, and G Zweig. 2013. word2vec. *https://code.google.com/p/word2vec* (2013).

Martin Mohr, Martin Hecker, Simon Bischof, and Johannes Bechberger. 2021. JOANA (Java Object-sensitive ANAlysis) - Information Flow Control Framework for Java. `https://pp.ipd.kit.edu/projects/joana`.

MyBatis. 2021. MyBatis: SQL mapper framework for Java. `http://www.mybatis.org/mybatis-3`.

NASA Ames Research Center. 2022. Java Pathfinder. `https://github.com/javapathfinder`.

Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2013. A Statistical Semantic Language Model for Source Code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 532–542. https://doi.org/10.1145/2491411.2491458

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

Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond J. Mooney. 2020. Learning to Update Natural Language Comments Based on Code Changes. arXiv:2004.12169 [cs.CL]

Andreas Prlić, Andrew Yates, Spencer E Bliven, Peter W Rose, Julius Jacobsen, Peter V Troshin, Mark Chapman, Jianjiong Gao, Chuan Hock Koh, Sylvain Foisy, et al. 2012. BioJava: an open-source framework for bioinformatics in 2012. *Bioinformatics* 28, 20 (2012), 2693–2695.

Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided Program Reasoning Using Bayesian Inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 722–735. https://doi.org/10.1145/3192366.3192417

Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 111–124. https://doi.org/10.1145/2676726.2677009

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 419–428. https://doi.org/10.1145/2594291.2594321

B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. ACM, New York, NY, USA, 12–27. https://doi.org/10.1145/73560.73562

Stuart J Russell and Peter Norvig. 2016. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Haşim Sak, Andrew Senior, and Françoise Beaufays. 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Fifteenth annual conference of the international speech communication association*.

F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks* 20, 1 (Jan. 2009), 61–80. https://doi.org/10.1109/TNN.2008.2005605

Adrian Smith. 2019. Universal Password Manager. http://upm.sourceforge.net.

A. Sureka and P. Jalote. 2010. Detecting Duplicate Bug Report Using Character N-Gram-Based Features. In *2010 Asia Pacific Software Engineering Conference*. 366–374. https://doi.org/10.1109/APSEC.2010.49

Susi.ai. 2018. api.susi.ai - Software and Rules for Personal Assistants. http://susi.ai.

The Apache Software Foundation. 2022. Apache Jackrabbit is a fully conforming implementation of the Content Repository for Java Technology API. http://jackrabbit.apache.org.

The Clang Team. 2021. Clang 12 documentation. https://releases.llvm.org/12.0.0/tools/clang/docs/index.html.

The HSQL Development Group. 2021. HyperSQL DataBase. http://hsqldb.org.

Chris Thunes. 2020. javalang: Pure Python Java parser and tools. https://pypi.org/project/javalang/, Accessed: 2022-02-13.

Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. 2014. ALETHEIA: Improving the Usability of Static Security Analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 762–774. https://doi.org/10.1145/2660267.2660339

Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the Localness of Software. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 269–280. https://doi.org/10.1145/2635868.2635875

Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multi-Modal Attention Network Learning for Semantic Source Code Retrieval. arXiv:1909.13516 [cs.SE]

Junjie Wang, Song Wang, and Qing Wang. 2018. 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* (Oulu, Finland) *(ESEM '18)*. ACM, New York, NY, USA, Article 17, 10 pages. https://doi.org/10.1145/3239235.3239523

Wenhua Wang, Yuqun Zhang, Zhengran Zeng, and Guandong Xu. 2020. TranS$\hat{3}$: A Transformer-based Framework for Unifying Code Summarization and Code Search. arXiv:2003.03238 [cs.SE]

Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 439–449.

Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep Learning Code Fragments for Code Clone Detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 87–98. https://doi.org/10.1145/2970276.2970326

Achilleas Xypolytos, Haiyun Xu, Barbara Vieira, and Amr MT Ali-Eldin. 2017. 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*. IEEE, 595–596.

Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. 2016. From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 404–415. https://doi.org/10.1145/2884781.2884862

U. Yüksel and H. Sözer. 2013. Automated Classification of Static Code Analysis Alerts: A Case Study. In *2013 IEEE International Conference on Software Maintenance*. 532–535.