



An empirical assessment of machine learning approaches for triaging reports of static analysis tools

Sai Yerramreddy¹ · Austin Mordahl² · Ugur Koc¹ · Shiyi Wei² · Jeffrey S. Foster³ · Marine Carpuat¹ · Adam A. Porter¹

Accepted: 31 October 2022 / Published online: 10 January 2023

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2023

Abstract

Despite their ability to detect critical bugs in software, static analysis tools' high false positive rates are a key barrier to their adoption in real-world settings. To improve the usability of these tools, researchers have recently begun to apply machine learning techniques to classify and filter incorrect analysis reports. Although initial results have been promising, the long-term potential and best practices for this line of research are unclear due to the lack of detailed, large-scale empirical evaluation. To partially address this knowledge gap, we present a comparative empirical study of three machine learning techniques—traditional models, recurrent neural networks (RNNs), and graph neural networks (GNNs)—for classifying correct and incorrect results in three static analysis tools—FindSecBugs, CBMC, and JBMC—using multiple datasets. These tools represent different techniques of static analysis, namely taint analysis and model-checking. We also introduce and evaluate new data preparation routines for RNNs and node representations for GNNs. We find that overall classification accuracy reaches a high of 80%–99% 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 5% for RNNs and 16% for GNNs. Overall, our results suggest that neural networks (RNNs or GNNs) that learn over a program's source code outperform traditional models, 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

Communicated by: Andrea De Lucia

Sai Yerramreddy and Austin Mordahl contributed equally to this research.

Sai Yerramreddy
saiyr@cs.umd.edu

Extended author information available on the last page of the article.

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). As a result, developers who use SA tools may spend significant time manually investigating reports (false positives), and important bugs or vulnerabilities could still be missed (false negatives). Many regard this issue as a key barrier to using SA tools in practice (John-son 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.

Several researchers have proposed using ML techniques with hand engineered features to classify and filter false positives (Heckman 2007, 2009; Yüksel and Sözer 2013; Tripp et al. 2014; Utture et al. 2022; Kang et al. 2022). 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 approaches (Yüksel and Sözer 2013; Tripp et al. 2014; Heckman 2009). Potential limitations of these approaches include that feature identification often relies on manual investigations by experts, and in cases where they have to be extracted from code, this process can be time-consuming. Furthermore, these approaches are not well-suited to represent 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 lead 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 dataset involved in that study, however, further study is called for.

Overall, while existing research shows potential benefits of using machine learning (ML) algorithms to classify SA results, there are still important open research questions. First and foremost, there has been relatively little empirical evaluation of different ML algorithms, which is of great practical importance for understanding their tradeoffs and requirements. 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—traditional models, recurrent neural networks and graph neural networks—for classifying SA results from three real-world tools (Section 2 overviews our study). These tools are FindSecBugs (Arteau et al. 2018), which implements a taint analysis for Java, and CBMC (Kroening and Tautschnig 2014) and JBMC (Cordeiro et al. 2018), which are model checkers for C and Java code, respectively (Section 3). Note that while these tools cover some major types of static analysis, such as information flow analysis and model checking, there are other types of static analysis we do not consider (Section 7.5). In our study, we collected multiple datasets, containing both real-world and artificial program sets. 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 2371 reports from 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 programs 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 4).

To accomplish our classification task, we experimented with three families of ML approaches: traditional models, recurrent neural networks, and graph neural networks (Section 5.1). We further experimented with several broad representations of the input data. First, we tried representing programs as hand-engineered feature vectors. Then, we tried different representations based on the form of the raw input data. Since FindSecBugs emits reports in the forms of flows from sources to sinks, for that tool we computed backward slices from the reported sink and used the corresponding program dependence graph (PDG) (Ferrante et al. 1987). For CBMC and JBMC, which may not emit a specific line we can slice from, we instead used the abstract syntax tree (AST) of the program (Section 5.2). We propose and implement multiple ways to use these representations as inputs to ML models, 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 5.3).

We compare the effectiveness of these three families of ML approaches with different combinations of data preparation routines. Our experimental results provide significant insight into the performance and applicability of the ML algorithms and data preparation techniques. First, we observe that neural networks perform better compared to the other approaches, with recurrent neural networks and graph neural networks performing the best at classifying program slices and ASTs, respectively. Second, with more precise data preparation, we achieve large performance improvements over the state-of-the-art (Koc et al. 2017), up to 89% accuracy on the real-world dataset. Third, we find that the data preparation for neural networks has a significant impact on the performance and generalizability of the approaches, and that different data preparation techniques should be applied in different application scenarios. Finally, we find that despite their overall inferior accuracy, there are still examples which only the traditional models were able to correctly classify (Section 7).

This work extends our previous work in ICST 2019 (Koc et al. 2019). In that work, we collected the OWASP and real-world (hereafter referred to as ICST) datasets. This work additionally includes the SV-COMP datasets for C and Java. In the previous work, we proposed and implemented various data preparation techniques to adapt the ML approaches we use to the task of triaging reports. These data preparation routines were based on different program representations, specifically hand-engineered features and a program slice represented by a subgraph of the program dependence graph (PDG). In this work, we adapt the data preparation techniques to new data and representations – specifically, we add the abstract syntax tree (AST) as a new program representation from which our models can learn. The results of this work confirm and expand upon those from our previous work: neural networks tend to perform best across all datasets, and data preparation has a large impact on neural network performance.

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 effectiveness of ML techniques on these tools, we augmented our dataset with 1368 new data points.

- 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 16%.

2 Overview

Recall that our goal is to study the performance of different machine learning models on triaging reports of different static analysis tools. Figure 1 summarizes the combinations of tools, datasets, data preparation routines, and ML models we study.

Our study focuses on static analysis tools for two target languages: C and Java (Column 1 of Fig. 1). We chose these two languages because there are various analyzers and datasets available for evaluation (Kroening and Tautschnig 2014; Cordeiro et al. 2018; Beyer 2018, 2019; Arteau et al. 2018; Burato et al. 2017). Column 2 shows the static analyzer(s) we used for each language: CBMC (Kroening and Tautschnig 2014) for C, and JBMC (Cordeiro et al. 2018) and FindSecBugs (Arteau et al. 2018) for Java. We selected these tools because they represent different program analysis approaches across different languages, allowing us to draw more general conclusions from our evaluation. We also selected them because they are actively maintained, well-known, and accompanied by datasets. These tools and the configurations under which we ran them are described in more detail in Section 3.

Column 3 shows the program sets we used for each tool. For both CBMC and JBMC, we used the SV-COMP program set (Beyer 2018, 2019), comprising both C and Java programs with ground truth data about the presence or absence of bugs. For FindSecBugs, we used two program sets. First, we used the popular OWASP program set (OWASP 2014). Second, we used the ICST dataset, which consists of manually classified analysis reports from real-world Java programs. We constructed this dataset in our preceding ICST work (Koc et al. 2019).

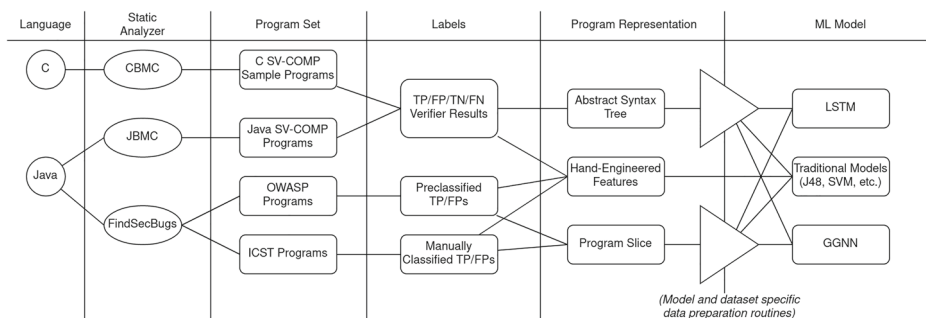


Fig. 1 Study Overview

Tool Output \ Ground Truth	Safe	Unsafe
	True negative	False negative
Safe		
Unsafe	False positive	True positive

Fig. 2 Labels for Training ML Models on CBMC and JBMC data

To train the ML models, we need to label each output of the tool as correct (i.e., a true positive/negative) or incorrect (i.e., a false positive/negative). We obtained these labels (Column 4 of Fig. 1) in three main ways. For CBMC and JBMC, we ran the tools on each program in the program set to get the tool's determination of the program's safety and compared that to the ground truth, which is included as part of the program set. Figure 2 shows how we obtained true/false positive/negative classifications for CBMC and JBMC. For FindSecBugs on OWASP, the tool output and labels were already available, so we used that data as is. For FindSecBugs on the ICST program set, we reused the manual classifications we presented in the previous work. Note that we have no true negative or false negative reports associated with either the OWASP or ICST datasets; this would require knowledge of all of the security issues in the target programs, which is unavailable. We describe the program sets and the obtained datasets in more detail in Section 4.

Column 6 shows the ML models we trained. We tried four discrete experiments, which are classified based on the representation of the dataset. First, we trained various traditional models, such as Naïve Bayes (Rish et al. 2001), decision trees (Safavian and Landgrebe 1991), and perceptrons (Rosenblatt 1958) on hand-engineered features (HEF) extracted from the data. Second, we treated the labeled data as bags of words (BoW), and trained J48 decision tree models (Quinlan 2014). We chose only J48 as it was the best performing model for HEF over all the datasets. Third, we trained a recurrent neural network known as LSTM (long short term memory) on a sequence-based representation of the data (Carrier and Cho 2018). Finally, we trained gated graph neural networks (GGNNs) on a graph representation of the data. We choose LSTM as our RNN implementation, since they are known to perform better on long sequences and in general provide better accuracy (Jozefowicz et al. 2015). We choose GGNN for our GNN experiments as it has achieved state-of-the-art results on problems from program verification (Li et al. 2015a) and has been designed for tasks where sequence and contextual information is important. In order to train these models, we used various different program representation techniques (Column 5 of Fig. 1). More details about the data preparation routines we used and the models themselves are discussed in Section 5.

3 Tools

In this section, we describe the SA tools we studied in more detail.

3.1 FindSecBugs

The first SA tool we study is FindSecBugs (Arteau et al. 2018), version 1.4.6. FindSecBugs is a static analyzer for Java programs that focuses on finding security issues. At the time

of writing, FindSecBugs detects 141 vulnerability patterns, such as using predictable pseudorandom number generators or property leaks. FindSecBugs is well-known and popular. As of July 2022, its GitHub repository had over 428 forks and 1900 stars (Arteau et al. 2018). FindSecBugs comes with various analyses; we configured it to do taint analysis, which is capable of detecting SQL, command, CRLF, and LDAP injections; cross-site scripting (XSS); and path traversal (XPATH) vulnerabilities. FindSecBugs emits alarms in both HTML and XML format. An example of the HTML format is shown in Fig. 3, which shows a list of potential vulnerabilities in the H2 program. Bug reports are associated with a type, a priority (indicated by the color in the left column), sources/sinks, and line numbers. For example, the report that is expanded in Fig. 3 indicates that there is a call to `Runtime.exec()` that is potentially vulnerable to command injection on line 455 of `FtpServer.java`. The alarm type is SECCI (indicating a SEcURITY Command Injection), and the red color indicates high priority. The report shows the path the tainted data takes from the source (line 62 of `FtpControl.java`) to the identified sink call (line 455 of `FtpServer.java`).

3.2 CBMC/JBMC

CBMC (Clarke et al. 2004) and JBMC (Cordeiro et al. 2018), the C/Java Bounded Checkers, can check various kinds of program properties, such as whether a program terminates or whether it contains buffer overflows. Specifically, these tools use *bounded model checking*. This is a technique in which a program is modeled as a finite state machine, and a property to verify is represented in terms of temporal logic. Then, given some integer bound k , the bounded model checker constructs a propositional formula that is satisfiable if and only if a counterexample (i.e., a series of program states that result in the property being violated) bounded in size by length k exists, and then uses a SAT solver to determine whether the formula is satisfiable (Biere et al. 1999).

Unlike FindSecBugs, CBMC and JBMC report that programs are either safe or unsafe with respect to the checked properties. Figure 4 shows an example of the output produced by running CBMC on a C file, including various assertions it generates and checks. For each assertion, SUCCESS indicates that the tool was able to prove the assertion true, and thus the

Code	Warning
SECCI	This usage of <code>ProcessBuilder.command(...)</code> can be vulnerable to Command Injection
SECCI	This usage of <code>Runtime.exec(...)</code> can be vulnerable to Command Injection
	Bug type COMMAND_INJECTION (click for details) In class <code>org.h2.dev.ftp.server.FtpServer</code> In method <code>org.h2.dev.ftp.server.FtpServer.startTask(String)</code> At <code>FtpServer.java</code> : [line 455] Sink method <code>Runtime.exec(...)</code> Method usage with tainted arguments detected At <code>FtpControl.java</code> : [line 62] At <code>FtpControl.java</code> : [line 69] At <code>FtpControl.java</code> : [line 85] At <code>FtpControl.java</code> : [line 102] At <code>FtpControl.java</code> : [line 311] At <code>FtpControl.java</code> : [line 318] At <code>FtpServer.java</code> : [line 455]
SECCI	This usage of <code>Runtime.exec(...)</code> can be vulnerable to Command Injection
SECCI	This usage of <code>Runtime.exec(...)</code> can be vulnerable to Command Injection
SECCI	This usage of <code>Runtime.exec(...)</code> can be vulnerable to Command Injection

Fig. 3 An example of an alarm reported by FindSecBugs

```

** Results:
[] free called for new[] object: SUCCESS
[_start.memory-leak.1] dynamically allocated memory
  never freed in __CPROVER_memory_leak == NULL: SUCCESS
[reach_error.assertion.1] assertion 0: SUCCESS
[free_data.assertion.1] free argument is dynamic object: SUCCESS
[free_data.assertion.2] free argument has offset zero: SUCCESS
[free_data.assertion.3] double free: SUCCESS
[free_data.assertion.4] free called for new[] object: SUCCESS
[free_data.assertion.5] free argument is dynamic object: SUCCESS
[free_data.assertion.6] free argument has offset zero: SUCCESS
[free_data.assertion.7] double free: SUCCESS
[free_data.assertion.8] free called for new[] object: SUCCESS

** 0 of 11 failed (1 iteration)
VERIFICATION SUCCESSFUL

```

Fig. 4 An example of CBMC's output

safety of the program with regard to that assertion (e.g., that memory allocated with `new[]` is later freed). We chose these two tools because they represent a different type of analysis than FindSecBugs, and they are popular and well known: as of July 2022, the original paper describing CBMC has over 1700 citations.

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). The default configurations of these tools are tuned to be both sound and precise, frequently failing to terminate rather than produce an incorrect result. For this study, 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. This is because the scenario we are evaluating is whether machine learning approaches can assist users in cases when tools emit many false results. We went through the sample configurations that Koc et al. (2021) generated from three-way covering arrays (Nie and Hareton 2011) in order to find such a configuration. For CBMC, we used 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`; for JBMC, we used the configuration `-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-nonnet-string-length 100 -max-nondet-tree-depth 100 -java-max-vla-length 10 -paths lifo -arrays-uf-never -reachability-slice -yices`.¹

¹This was one of 24 configurations of JBMC that produced the exact same distribution of correct/incorrect results on our dataset (Section 4).

Table 1 A summary of the datasets we used

Dataset	Languages	Tools	# Datapoints (T/F)	Label type
OWASP	Java	FindSecBugs	1124/1193	Bug Reports
ICST	Java	FindSecBugs	194/206	
JBMC	Java	JBMC	164/204	Verification Results
CBMC	C	CBMC	483/517	

4 Datasets

As discussed in Section 2, we used four datasets in this work (three Java and one C) which are summarized in Table 1. These datasets include both real-world programs and synthetic programs. In our previous work, we generated two of these datasets, comprising classified FindSecBugs reports, including reports from both real-world and synthetic program sets. In this work, we augmented this collection with two new datasets, consisting of true/false positive/negative results from JBMC and CBMC.

The first dataset we constructed in our previous work was derived from the OWASP Benchmark (OWASP 2014), which has been used to evaluate various SA tools in the literature (Koc et al. 2017; Burato et al. 2017; Xypolytos et al. 2017). OWASP consists of thousands of Java test cases with various known vulnerabilities, each mapped to CVEs. In particular, we used the same programs as Koc et al. (2017) so we could compare results. Our dataset contains 2 371 FindSecBugs SQL injection vulnerability reports, 1 193 of which are labeled as false positives; the remaining reports are labeled as true positives.

The second dataset of FindSecBugs reports was classified from a program set of 14 real-world Java programs. Since this is a program set we constructed in our original publication, we call this the ICST program set. We ran FindSecBugs on these programs and then manually labeled the resulting vulnerability reports as true or false positives. We chose these programs using the following criteria:

- We selected programs for which FindSecBugs generates *vulnerability reports*. To have the kinds of vulnerabilities we study, we observe that programs should perform database and LDAP queries, use network connections, read/write files, and/or execute commands.
- We chose programs that are *open source* because we need access to source code to apply our ML algorithms.
- We chose programs that are under *active development* and are *highly used*.
- Finally, we chose programs that are *small to medium size*, ranging from 5K to 1M lines of code (LoC). Restricting code size was necessary to successfully create the PDG, which is used for program slicing (Mohr et al. 2021).

Table 2 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 HSQLDB, 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 5 363 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 400 of these 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 using Eclipse's built-in support for constructing call hierarchies from the method that has the reported error line (Eclipse Foundation 2022a). Then we perform a depth-first search on the backward call tree, inspecting 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. This process was done manually, using the labeler's best judgment in terms of determining whether the data produced by a source was actually tainted and whether tainted data was ever sanitized or checked for safety. One author performed most of the labeling work, then selected about 5% of the classified results to present to the rest

Table 2 ICST program set

Program	Description	LoC	# reports	
			TP	FP
Apollo-0.9.1	distributed config. (Apollo 2018 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 2022b)	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)	6 358	2	13
Susi.AI-07260c1	artificial intel. API (Susi.ai 2018)	65 388	47	46
Total		—	194	206

of the authors. This presentation included the detailed step-by-step process of obtaining each classification in the sample. The rest of the authors observed and validated this process.

Through this review process, we observed that the false positives we found in the ICST dataset were significantly different from those in the OWASP 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 dataset, false positives mostly stem from the first scenario. In our ICST dataset, we mostly see only the second and third scenarios. This demonstrates the importance of creating a real-world dataset for our study.

This work contributes two new datasets, which are based on 1368 programs from the annual 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 an aggregation of multiple independent artificial benchmarks from various contributors, and is, to our knowledge, the largest single collection of verification tasks for C and Java for which the ground truths are known. Each program has associated ground truth, such that with regard to some property (e.g., memory safety), the program is known to be safe or unsafe. We used all 368 Java programs 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 SAT-UNE (Koc et al. 2021). We sampled C programs from SV-COMP because running multiple configurations on all programs in the corpus (i.e., more than 10000 programs) in order to identify configurations that gave a balanced configuration would be prohibitively expensive. In addition, the sample we took only consisted of programs that had one ground truth associated with them. SV-COMP has programs that have multiple ground truths for different program verification properties, which would introduce noise into the dataset if we include duplicate data points with different classifications. Since the ground truths are already known from the dataset, we did not perform manual classification as was necessary for the FindSecBugs' ICST dataset. Instead, we ran the configurations described in Section 3.2 and compared the tools' results to the ground truth to obtain labels. CBMC produced 457 incorrect results (i.e., false positives or false negatives), 523 correct results (i.e., true positives or true negatives), and 20 inconclusive results (i.e., where the tool did not complete analysis within 1 minute). We discarded the 20 inconclusive results. JBMC produced 204 incorrect results and 164 correct results, with no inconclusive results. For convenience, we hereafter refer to the subset of the SV-COMP programs for C as the CBMC dataset, and the subset of the SV-COMP programs for Java as the JBMC dataset.

5 ML Models and Data Representations

In this section, we describe the three families of models we studied, along with how we represented and encoded programs as inputs to the machine learning models.

5.1 ML Models

Our goal in this work is to assess their strengths and weaknesses of different machine learning models with regard to the problem of classifying static analysis results. In this subsection, we provide background on the three families of approaches we studied, as shown in the last column of Fig. 1.

5.1.1 Traditional Models

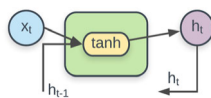
We frame the detection of incorrect results as a standard binary classification problem (Russell and Norvig 2016). Given an input x e.g., a point in a high dimensional space R^D , the classifier produces an output $y = f_{\theta}(x)$, where $y = 1$ for an incorrect result (false positive or false negative), and $y = 0$ otherwise. Constructing such a classifier requires defining an input vector 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 $\{(x_1, y_1) \dots (x_N, y_N)\}$.

We experimented with various instantiations of traditional machine learning models, specifically naïve Bayes, Bayesian networks, decision trees, random forests, multi-layer perceptrons, support vector machines, and clustering. These classifiers use relatively simple (compared to neural nets) models of the data and tend to produce highly interpretable results. For example, naïve Bayes assumes that all independent variables are statistically independent from each other² and thus computes the probability of an example with features $x_1, \dots, x_n$ having a label Y_i being directly proportional to $(Y_i) \prod_{j=1}^n p(x_j | Y_i)$ per Bayes' rule.³

5.1.2 Recurrent Neural Networks

For text classification, recurrent neural networks (RNNs) (Mandic and Chambers 2001; Hochreiter and Schmidhuber 1997; Gers et al. 2000) 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 $X = x_0, x_1, \dots, x_t, \dots, x_n$ from left to right, one position at a time. For each position t , RNNs compute a 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 $h_0, h_1, \dots, h_n$ 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

²Two events A and B are statistically independent iff $P(A \cap B) = P(A)P(B)$.

³ $p(x|y) = (p(y)p(y|x))/p(x)$

prefix $x_0, \dots, x_{t-1}$ 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 traditional models.

Researchers have used RNNs to solve software engineering tasks such as code completion (Dam et al. 2016) and code synthesis (Kushman and Barzilay 2013; Ling et al. 2016). Specifically, Koc et al. (2017) conducted a case study using Long Short-term Memories (LSTM) (Hochreiter and Schmidhuber 1997; Gers et al. 2000), a popular kind of RNN, for classifying SA false positives. In this work, we study LSTMs as well as they are commonly applied to NLP programs. Our problem can be treated as an NLP task because 1) our data is sequential, and 2) it has contextual (long-term) dependencies (Sak et al. 2014) which could be relevant to classify an SA result as correct or incorrect.

5.1.3 Graph Neural Networks

With RNNs, we represent a program 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 $G = (N, E)$, where $N = \{n_0, n_1, \dots, n_i\}$ is the set of nodes, and $E = \{e_1, e_2, \dots, e_j\}$ 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 $e = (type, source, dest)$, 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 $NBR(n_i)$ at each time step t , i.e., $h^{(t)} = f(n_i, h^{(t-1)}, NBR(n_i))$. The function f is defined as a neural network.

Programs can intuitively be represented as graphs (e.g., as an AST) so exploring GNNs makes sense. In our study, we focus on a variation of GNNs called Gated Graph Neural Networks (GGNN) (Li et al. 2015b). GGNNs have gated recurrent units and enable initialization of the node representation. We chose GGNNs because they have achieved high accuracy on problems from program verification Li et al. (2015a). GGNNs have been used to learn properties about programs (Li et al. 2015b; Allamanis et al. 2017), but, to our knowledge, have not been applied to classify SA results outside of our previous work.

5.2 Representing Programs

Before we use the source code/file for the approaches mentioned in Section 5.1, we must transform them into representations that can be interpreted by the models. We use two different program representations, one for the programs analyzed by FindSecBugs, and one for the programs analyzed by CBMC and J BMC.

Each report generated by FindSecBugs contains a source line, indicating the location of the bug. For example, Fig. 3 shows the source line as being line 455 of FtpServer.java. We can use this information to narrow down the code to the most relevant parts 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

```

1  EXPR 164 {
2    0 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. 5 Sample PDG node (simplified for presentation)

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 program set, which is composed of large, real-world programs (see Table 2).

This approach works for FindSecBugs reports, but not for CBMC and JBMC reports, as the latter do not always include a source line, as can be seen in Fig. 4. For example, termination checks do not emit a faulty line number if the program is determined not to terminate. Thus, without a consistent slicing criterion for these tools' reports, we instead use the full source code as input, represented as an AST.

5.2.1 Representing Programs as Slices

We computed backwards slices for FindSecBugs results using 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 nodes denoting the program locations that are reachable from the entry points, and edges denoting control- or data-flow dependencies between nodes. 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 5 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.⁴ 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.

⁴Joana uses the intermediate representation from the T.J. Watson Libraries for Analysis (*WALA*) (IBM 2006).

5.2.2 Representing Programs as ASTs

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 6 shows an example of an AST (Fig. 6b) generated for a piece of code (Fig. 6a). On line 1 in Fig. 6b, we see the target variable which denotes the program label. The value $[0, 1]$ denotes an incorrect result, and the value $[1, 0]$ denotes a correct result. On line 2, we see the entire AST graph structure represented using edges. $[0, 1]$ represents an edge from node 0 to node 1. On line 3, we see the Node Type for all the nodes present in the AST. For example, node 0 is of the type `CompilationUnit`. On 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;`

```

1 import org.sosy_lab.sv_benchmarks.Verifier;
2
3 public class Main {
4     public static void main(String[] arg) {
5         int i = 0;
6         boolean b = Verifier.nondetBoolean();
7
8         while (true) {
9             i++;
10            assert (b);
11        }
12    }
13 }

```

(a) A Java program from the JBMC dataset.

```

1 targets: [[0, 1]],
2 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]],
3 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"],
4 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. 6 An example of the AST generated for a Java program in the JBMC dataset

5.3 Data Preparation Routines

We now detail the specific ways we prepared the data for each model, as shown between Columns 5 and 6 of Fig. 1.

5.3.1 Traditional Models

Hand-Engineered Features A feature vector 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 x . Indeed, researchers have used this approach to classify SA false positives in prior work (Heckman 2007, 2009; Yüksel and Sözer 2013; Tripp et al. 2014; Utture et al. 2022; Kang et al. 2022).

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 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.

For all programs, we extracted some common features from the source code, specifically the number of path conditions and the number of function calls. These features have been used by other researchers (Tripp et al. 2014; Utture et al. 2022; Kang et al. 2022). We then used different methods for the rest of the feature vector depending on whether we were generating data from FindSecBugs or CBMC/JBMC.

Since FindSecBugs reports include a wealth of information about the detected bug and the code responsible, we primarily used these reports as the source of features. Specifically, we adapted the approach by 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) *rule name*, and (12) *severity*. The first seven features are lexical, the next three are quantitative, and last two are security-specific. These are in addition to the common features noted above. 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. 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 to 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.

For programs analyzed by CBMC and JBMC, we adopted Koc et al.'s approach for feature extraction, which they performed on the same SV-COMP datasets (Koc et al. 2021). We converted the source code of C and Java programs 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 constructs: (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,⁵ (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 Figs. 7 and 8 respectively. In each figure, the first subfigure shows code and the second subfigure shows the HEF extracted from that code. For example, Fig. 7b shows a value of 3 for the feature numLoops, referring to the loops on lines 10, 14, and 20 respectively.

HEF can be regarded as the state-of-the-practice approach for this problem (Wang et al. 2018). However, as the input is often manually condensed into a discrete set of features using human expertise, this approach often cannot comprehend the deep structure of the source code being analyzed. Next, we explore how to represent program source code for more complex ML approaches that can implicitly learn feature representations.

Bag of Words One intuitive approach to learn from data is to transform it directly into a feature vector. 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 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.

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 5.3.2 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.

⁵Number of composite features include counting the number of variables, ifs, loops, functions defined, functions called, loads, and stores.

```

1 extern void __VERIFIER_error() __attribute__((__noreturn__));
2 void __VERIFIER_assert(int cond) { if(!(cond)) { ERROR: __VERIFIER_error(); } }
3 extern int __VERIFIER_nondet_int();
4 #define size 10000
5 int main()
6 { int a[size];
7   int b[size];
8   int i = 0;
9   int j = 0;
10  while( i < size )
11  { b[i] = __VERIFIER_nondet_int();
12    i = i+1; }
13  i = 0;
14  while( i < size )
15  {   a[j] = b[i];
16      i = i+1;
17      j = j+1; }
18  i = 0;
19  j = 0;
20  while( i < size )
21  {   __VERIFIER_assert( a[j] == b[j] );
22      i = i+1;
23      j = j+1; }
24  return 0; }

```

(a) A C program from CBMC dataset.

```

1 filename: array-examples/standard_copy1_true-unreach-call_ground.i
2 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, numFPTToSI: 0, numFPTToUI: 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,
3 Label: TrueResult

```

(b) The corresponding HEF features.

Fig. 7 An example of the HEF generated for a C program in the CBMC dataset

5.3.2 LSTM

To use LSTM, we need to transform program representations into sequences of tokens, which we achieve with four sets of transformations. To the best of our knowledge, the effects of such 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, splitting paths of classes and methods, and removing certain nodes to improve the effectiveness and generalizability.

```

1 import org.sosy_lab.sv_benchmarks.Verifier;
2
3 public class Main {
4     public static void main(String[] arg) {
5         int i = 0;
6         boolean b = Verifier.nondetBoolean();
7
8         while (true) {
9             i++;
10            assert (b);
11        }
12    }
13 }

```

(a) A Java program from JBMC dataset.

```

1 entryClass: jayhorn-recursive/InfiniteLoop
2 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,
3 Label: FalseResult

```

(b) The corresponding HEF features.

Fig. 8 An example of the HEF generated for a Java program in the JBMC dataset

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, as shown in Table 3. All but one of these transformations were applied regardless of whether the data was a slice or an AST. One transformation, T_{cIn} , was applied only to slices.

Common Transformations

- *Data Tokenization (T_{tkn})* We tokenize programs, converting the content of each graph node into a sequence of tokens. On PDGs, we implemented our own tokenization routine, which performs some cleanup (e.g., removing the dot between a receiver and a method call, or removing the preceding L from class references in bytecode). 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})* 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.

Table 3 Preparations, their names, and the datasets to which they were applied. Note that the * in *HEF*-* denotes the specific model trained, which are enumerated in Section 5.1

Applied preparations	Approach name	Datasets
Hand-engineered feature extraction	<i>HEF</i> -*	OWASP, ICST CBMC, JBMC
Occurrence feature vector	<i>BoW-Occ</i>	
Frequency feature vector	<i>BoW-Freq</i>	
T_{cln}	<i>LSTM-Raw</i>	OWASP, ICST
$T_{cln} + T_{ans}$	<i>LSTM-ANS</i>	
$T_{cln} + T_{ans} + T_{aps}$	<i>LSTM-APS</i>	
$T_{cln} + T_{ans} + T_{aps} + T_{ext}$	<i>LSTM-Ext</i>	
Kind, operation, and type node vector	<i>GGNN-KOT</i>	
KOT + an extracted item	<i>GGNN-KOTI</i>	
Node Encoding	<i>GGNN-Enc</i>	
No preparation	<i>LSTM-Raw</i>	CBMC, JBMC
T_{ans}	<i>LSTM-ANS</i>	
$T_{ans} + T_{aps}$	<i>LSTM-APS</i>	
$T_{ans} + T_{aps} + T_{ext}$	<i>LSTM-Ext</i>	
Node Type	<i>GGNN-T</i>	
Node type + first N node content tokens	<i>GGNN-NT</i>	
Node type + node content word encoding	<i>GGNN-EncT</i>	

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

Slice-Specific Transformation

- *Data Cleansing (T_{cln})* This set of transformations removes certain PDG nodes and performs basic tokenization. First, the transformations remove nodes of certain kinds

(i.e., `formalin`, `formalout`, `actualin`, `actualout`), 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.⁶ 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 do not perform this operation for CBMC and JBMC as we create ASTs for these tools and ASTs contain no redundant information.

5.3.3 GGNN

To adapt GGNNs to our problem, we explore several different node representations. We group the representations based on what type of information from the respective node (PDG or AST) we encode.

Enumerated Fields only (KOT/T) As the first representation, we only use enumerated fields. These are fields whose potential values are taken from a relatively limited set of potential values. For PDGs, these include the `Kind`, `Operation`, and `Type` fields of the graph nodes (KOT). For the example node in Fig. 5, the KOT node representation is $V_{rep} = [\text{EXPR}, \text{reference}, \text{Ljava/sql/Connection}]$. In ASTs, we only use the `Type` (T) fields of the graph nodes. For the example AST in Fig. 6b, for the first node, the node representation is $V_{rep} = [\text{Encoder}(\text{Node_Type}[0])]$ i.e., $[\text{Encoder}(\text{CompilationUnit})]$, where `Encoder` is used to Encode Node Type features using an ordinal encoding scheme. We refer to this encoding as *T* for PDGs and ASTs, respectively.

Enumerated Fields and Abstraction of Content (KOTI/NT) In the second representation, in addition to enumerated fields, we include some abstraction of the node's contents. In PDGs, 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. This representation is called *KOTI*. The KOTI representation for the PDG node in Fig. 5 is $V_{rep} = [\text{EXPR}, \text{reference}, \text{Ljava/sql/Connection}, \text{object}]$ (object is the abstraction of the extracted item `com.mangrove.utils.DBHelper.conn`, meaning that the reference is for an object).

For ASTs, in addition to the `Type` field, we tokenize and encode the tokens derived from the `NodeContent` field of the AST for a particular node. This is called the *NT* representation, with the *N* attribute in the name being the number of tokens being considered for

each node. The operation can be represented as $V_{rep} = E_T \# \sum_{i=0}^N \text{Encode}(x_i)$, where $\#$,

are the concatenation operations and x_i represent tokens for a particular node's content. transformations (ANS, APS, EXT) as aforementioned in the LSTM section. If a particular node has k tokens, where $k < N$, we concatenate $[-1] \boxtimes N - k$ times to V_{rep} , thus ensuring a constant size of $N+1$ for node representation.⁷ For the example AST in Fig. 6b, for

⁶A 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).

⁷ $[-1]$ here refers to an array with a single -1 element, if $N = 3$ and $k = 1$ then $[-1, -1]$.

the first node the `Node_Content` is empty, so the node representation with $N = 2$ is $V_{rep} = [\text{Encoder}(\text{Node_Type}[0]), -1, -1]$, whereas for the node with the `Node_Content` as “public static” the node representation is $V_{rep} = [\text{Encoder}(\text{Node_Type}[0]), \text{LabelEncoder}(["public"]), \text{LabelEncoder}(["static"])]$.

Enumerated Fields and Encoding of All Node Content (Enc/EncT) In the third representation, we use word embeddings to compute a vector representation that accounts for the entire content of each node. In PDGs, this content is the bytecode in the `Value` field (which has an arbitrary number of words). To achieve this encoding, we first perform pre-training using a bigger, unlabeled dataset (created using tokens made during the LSTM step) to learn embedding vectors that capture more generic aspects of the words in the dictionary using the *word2vec* model (Mikolov et al. 2013; Goldberg and Levy 2014). 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.

For ASTs, we compute a vector representation (E_V) for each node using word embeddings from a *word2vec* model (Mikolov et al. 2013; Goldberg and Levy 2014) for all the tokens present in that node’s content `NodeContent`. The operation can be represented as

$$E_V = \frac{1}{N} \sum_{i=0}^N \text{Word2Vec}(x_i),$$

where $\sum$ 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.

6 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 three families of ML approaches described in Section 5. We split the traditional models into two categories based on whether we represented the input data as HEF or BoW (hereafter, we broadly refer to four categories of models: HEF, BoW, LSTM, and GGNN). For learning with HEF, we experimented with 9 classification algorithms: Naïve Bayes, Bayesian Network, Decision Tree (J48), Random Forest, Multi-layer Perceptron (MLP), K*, OneR, ZeroR, and support vector machines, with the features described in Section 5.3.1. We used the WEKA (Eibe et al. 2016) implementations of these algorithms.

For traditional models learning on BoW, RNN models, and GNN models, we experimented with the variations described in Sections 5.3.1, 5.3.2, and 5.3.3. Table 3 lists these variations with their names and the data preparation applied for them. For example, the approach *LSTM-Raw* on the OWASP and ICST datasets uses only the T_{cln} transformation alone, while *LSTM-Ext* uses all four transformations. For BoW, we only used Decision-Tree (J48) based on its good performance on HEF representations. For BoW and HEF, we also used the AutoML package (AutoML 2022) for hyper-parameter tuning. We adapted the LSTM implementation designed by Carrier and Cho (2018) by modifying the input layer to accept our data and changing the optimization function from ADADELTA (Zeiler

2012) to Adam (Kingma and Adam 2014), we do this to improve model performance on our classification tasks. We also extended the GGNN implementation from Microsoft Research (Microsoft 2019) by modifying the input layer and changing the output layer to return classifications rather than probabilities.

As shown in Table 3, 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, BoW as well as the HEF 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 datasets 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 dataset is not appropriate for the second scenario as all the programs in the dataset were developed by same people and hence share many common properties such as variable names, length, and API usage. We refer to the ICST program-wise split dataset as ICST-PW for short. The CBMC and JBMC datasets fit into the second scenario. Since each program appears once in these datasets, the training and the test set of programs are mutually exclusive.

Training Configuration Table 4 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 (1234, 3252, 5827, 7421, 9876). The purpose of these many repetitions is to evaluate whether the results are consistent (see Section 7.3).

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 ended training before the timeout if there was no accuracy improvement for a certain number of epochs; this cut-off epoch value is known as the *epoch patience*. For both the OWASP and ICST datasets, we set the epoch patience to 100 for GGNN and 20 for LSTM. For the CBMC dataset, we set the epoch patience to 150 and 100 for GGNN and LSTM. For the JBMC dataset, we set the epoch patience to 100 and 50 for GGNN and LSTM. For LSTM, we conducted small-scale preliminary experiments of 15 epochs with *LSTM-Ext* to determine the word embedding dimension for tokens and batch size. We tested 4, 8, 12, 16, 20, and 50 for the word 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 for CBMC and JBMC benchmarks. For the FindSecBugs benchmarks (OWASP, ICST-Rand, and ICST-PW), word embedding dimension 8 and batch size 8 performed best in the preliminary

Table 4 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

experiments. We use the Adam (Kingma and Adam 2014) optimizer and hyperbolic tangent function (tanh) as the activation function for the LSTM and GNN experiments. We used the embedding dimension of 12 (size of word2vec embeddings) for the pre-training of *GGNN-Enc*. We use dimension size 1 for encoding Tokens in *GGNN-NT*, as we use a simple label encoder to encode the tokens. 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. We use Gated Recurrent Unit (GRU) (Cho et al. 2014) as graph cells and tanh as the graph RNN activation function for the GNN experiments.

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:

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

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 datasets 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.

Implementation The experiments were conducted using scripts written in Python, Java and Bash. Most parts of the data preparation routines were developed using Python and generic Python libraries. AST generation was done using Clang version 12.0.0 (The Clang Team 2021) for CBMC and ASTExtractor version 0.5 (Diamantopoulos 2020) for JBMC. Specifically, the LSTM data preparation was implemented in Python with 210 lines of code (LoC) for CBMC and JBMC, and 183 LoC for FindSecBugs. The BoW data preparation was implemented in Python with 80 LoC for all the target tools. The GGNN data preparation was implemented in Python with 512 LoC for CBMC, 290 LoC for JBMC, and 560 LoC for FindSecBugs. The HEF and BoW models were developed using WEKA (Eibe et al. 2016); we used the WEKA GUI for tuning these models. The LSTM architecture was developed using Python 2.7 along with the Theano library (Theano Development Team 2016) (651 LoC), while the GGNN model architecture was developed based on an implementation by Microsoft Research (2019) using Python 3 and the Tensorflow library (Abadi et al. 2015) (795 LoC). More details about how each specific process was automated are present in our replication repository at <https://bitbucket.org/SaiArrow/emse-replication-package/>.

7 Analysis of Results

As seen in Table 4, we trained 2 350 SA report classification models in total. The summary of the results can be found in Tables 5 and 6, 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.

7.1 RQ1: Overall Performance Comparison

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

In Table 5, 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 consistently outperform other models 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 (i.e., ICST-PW), 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 7.4). The GGNN variations have a large difference in accuracy. *GGNN-Enc* achieves 94%, while the other two variations achieve around 80% accuracy.

Table 5 Test recall, test precision, test accuracy and train accuracy results for the approaches in Table 3 and four most accurate algorithms for HEF, sorted by test 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	Train accuracy
OWASP	<i>LSTM-Raw</i>	100.00 ₀	100.00 ₀	100.00 ₀	100.00 ₀
	<i>LSTM-ANS</i>	99.15 _{0.74}	98.74 _{0.42}	99.37 _{0.42}	100.00 ₀
	<i>LSTM-Ext</i>	98.94 _{1.90}	99.57 _{0.44}	99.16 _{1.16}	100.00 ₀
	<i>LSTM-APS</i>	98.30 _{0.42}	99.14 _{0.21}	98.53 _{0.27}	100.00 ₀
	<i>BoW-Occ</i>	97.90 _{0.45}	97.90 _{1.25}	97.47 _{0.74}	99.40 _{0.30}
	<i>BoW-Freq</i>	97.90 _{0.45}	97.00 _{0.25}	97.26 _{0.31}	99.40 _{0.05}
	<i>GGNN-Enc</i>	92.00 _{5.00}	94.00 _{5.25}	94.00 _{1.60}	96.50 _{1.50}
	<i>HEF-J48</i>	88.50 _{1.65}	75.10 _{0.50}	79.96 _{0.21}	82.20 _{0.15}
	<i>GGNN-KOTI</i>	78.50 _{6.25}	81.00 _{2.50}	79.00 _{1.95}	78.50 _{0.15}
	<i>HEF-RandomForest</i>	85.50 _{1.65}	74.10 _{0.65}	78.32 _{0.50}	86.30 _{0.15}
	<i>GGNN-KOT</i>	80.00 _{3.25}	77.50 _{2.00}	78.00 _{0.95}	79.65 _{0.35}
	<i>HEF-K*</i>	84.70 _{2.05}	73.60 _{0.90}	77.68 _{1.37}	85.60 _{0.10}
	<i>HEF-MLP</i>	79.10 _{7.00}	70.90 _{2.10}	73.00 _{1.27}	79.60 _{0.90}
ICST-Rand	<i>LSTM-Raw</i>	90.62 _{2.09}	86.49 _{3.52}	89.33 _{2.19}	96.31 _{2.46}
	<i>LSTM-Ext</i>	90.62 _{4.41}	85.29 _{3.20}	89.04 _{1.90}	96.00 _{2.16}
	<i>LSTM-APS</i>	91.43 _{4.02}	86.11 _{3.99}	87.67 _{2.85}	95.08 _{3.20}
	<i>LSTM-ANS</i>	89.29 _{2.86}	84.21 _{3.97}	87.67 _{1.59}	95.08 _{2.25}
	<i>BoW-Freq</i>	86.10 _{2.30}	87.90 _{1.85}	87.14 _{1.85}	96.60 _{0.95}
	<i>BoW-Occ</i>	84.40 _{4.45}	87.50 _{3.85}	85.53 _{2.45}	96.00 _{0.65}
	<i>GGNN-KOTI</i>	83.00 _{4.50}	84.00 _{3.50}	84.21 _{1.55}	89.50 _{2.65}
	<i>HEF-K*</i>	80.00 _{3.95}	85.70 _{2.30}	84.00 _{0.89}	98.80 ₀
	<i>HEF-RandomForest</i>	75.00 _{1.40}	84.40 _{3.20}	84.00 _{0.93}	99.70 _{0.15}
	<i>GGNN-KOT</i>	89.00 _{7.00}	80.00 _{7.00}	83.56 _{3.48}	87.75 _{6.50}
	<i>GGNN-Enc</i>	80.00 _{6.00}	78.00 _{4.50}	82.19 _{3.63}	86.65 _{4.50}
	<i>HEF-J48</i>	78.10 _{2.15}	82.40 _{0.90}	81.33 _{0.92}	90.20 _{2.05}
	<i>HEF-MLP</i>	71.40 _{2.80}	86.20 _{6.10}	81.33 _{1.97}	88.80 _{0.65}
ICST-PW	<i>LSTM-Ext</i>	78.57 _{12.02}	76.19 _{5.20}	80.00 _{4.00}	93.46 _{3.27}
	<i>LSTM-APS</i>	70.27 _{14.59}	76.47 _{6.70}	78.48 _{3.33}	90.12 _{6.72}
	<i>LSTM-ANS</i>	62.16 _{25.58}	75.76 _{7.02}	74.68 _{3.85}	95.08 _{6.54}
	<i>LSTM-Raw</i>	67.57 _{31.91}	79.66 _{8.40}	74.67 _{4.08}	96.00 _{5.12}
	<i>GGNN-Enc</i>	77.00 _{36.00}	75.00 _{19.50}	74.67 _{5.89}	89.34 _{12.68}
	<i>GGNN-KOT</i>	77.00 _{29.50}	72.00 _{16.25}	74.00 _{5.84}	86.50 _{17.50}
	<i>HEF-MLP</i>	58.10 _{14.65}	70.40 _{9.40}	73.08 _{7.76}	89.20 _{1.30}
	<i>GGNN-KOTI</i>	65.00 _{33.50}	75.00 _{11.00}	72.02 _{5.12}	83.50 _{10.90}
	<i>HEF-K*</i>	66.10 _{24.50}	60.60 _{14.90}	68.00 _{9.75}	98.40 _{0.35}
	<i>HEF-J48</i>	60.70 _{11.65}	72.70 _{12.80}	65.33 _{8.04}	89.20 _{3.60}
	<i>HEF-RandomForest</i>	62.50 _{24.30}	60.30 _{5.55}	63.44 _{2.67}	99.70 ₀
	<i>BoW-Occ</i>	50.00 _{12.90}	65.00 _{22.30}	51.32 _{4.61}	96.00 _{0.30}
	<i>BoW-Freq</i>	47.80 _{16.50}	65.70 _{14.70}	51.25 _{8.55}	97.60 _{0.20}

Table 5 (continued)

CBMC	<i>GGNN-NT-16</i>	86.60	0.85	80.20	1.25	83.40	0.75	84.50	0.50
	<i>LSTM-ANS</i>	83.00	1.02	79.74	0.42	81.37	0.74	86.40	1.57
	<i>LSTM-APS</i>	83.19	1.16	79.57	0.44	81.36	0.90	85.65	1.35
	<i>LSTM-Ext</i>	82.85	0.87	79.55	0.50	81.25	0.42	84.60	2.54
	<i>LSTM-Raw</i>	82.60	1.27	79.10	1.21	81.13	1.42	83.41	3.12
	<i>GGNN-NT-8</i>	83.60	1.55	77.60	1.67	80.60	1.25	82.40	1.5
	<i>HEF-RandomForest</i>	79.60	1.40	80.40	1.85	79.60	1.40	93.30	0.50
	<i>BoW-Occ</i>	79.20	1.15	79.20	1.65	79.20	1.15	84.80	0.15
	<i>BoW-Freq</i>	80.40	0.25	78.80	0.95	79.20	0.95	90.30	0.65
	<i>HEF-J48</i>	79.20	0.40	80.30	0.40	79.20	0.30	86.70	0.60
	<i>HEF-K*</i>	78.4	1.10	79.00	1.15	78.40	1.1	92.50	0.60
	<i>HEF-MLP</i>	78.40	1.00	80.60	0.95	78.40	1.00	80.70	0.50
	<i>GGNN-NT-5</i>	79.40	1.20	78.00	0.65	78.20	0.65	80.40	1.75
	<i>GGNN-EncT</i>	76.40	0.75	76.00	0.50	76.20	0.25	75.80	0.75
	<i>GGNN-T</i>	76.20	1.20	75.40	0.75	75.80	0.50	75.60	0.40
JBMC	<i>GGNN-NT-5</i>	82.10	3.05	77.60	3.50	80.15	3.30	84.25	2.50
	<i>GGNN-NT-2</i>	81.50	3.00	77.50	3.50	79.50	3.23	82.00	3.25
	<i>LSTM-APS</i>	78.60	5.24	77.97	2.23	78.50	2.39	85.08	2.20
	<i>GGNN-NT-1</i>	81.00	1.55	75.00	7.50	78.00	2.25	81.25	1.75
	<i>LSTM-Ext</i>	78.68	6.25	76.45	2.41	77.72	1.71	84.35	3.60
	<i>LSTM-ANS</i>	78.44	5.33	76.45	3.20	77.44	3.00	84.07	2.33
	<i>LSTM-Raw</i>	78.68	11.22	76.98	2.36	77.44	3.06	85.42	2.04
	<i>BoW-Freq</i>	73.70	1.35	75.70	1.40	74.70	1.35	90.50	0.35
	<i>BoW-Occ</i>	74.30	2.20	74.40	2.45	74.30	2.2	86.80	2.40
	<i>HEF-J48</i>	74.00	2.35	74.00	2.45	74.00	2.35	86.70	1.50
	<i>GGNN-EncT</i>	72.20	2.55	75.90	3.50	73.95	3.30	74.25	2.35
	<i>GGNN-T</i>	72.00	2.45	75.90	3.50	73.75	3.20	74.50	3.65
	<i>HEF-RandomForest</i>	73.00	4.4	73.80	4.15	73.00	4.4	94.20	0.50
	<i>HEF-MLP</i>	64.90	1.85	66.70	2.95	64.90	1.85	71.90	3.55

This suggests that for the OWASP dataset, the values 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 2 ICST dataset splits (ICST-Rand and ICST-PW), we suspect that the models overfit due to a comparatively large difference between train and test accuracy. This is unsurprising as the ICST dataset has only 400 datapoints. We observe that the ICST-PW experiments seems to show a lot of overfitting, which is again expected due to the difficulty of the application scenario. We further expand on this in Section 7.5. Our observation for the 2 ICST benchmarks are as follows: 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

Table 6 Number of epochs and training times for the LSTM and GGNN approaches. Median and SIQR values are shown as in Table 5

Dataset	Model	# of epochs	Training time(min)
OWASP	<i>LSTM-Raw</i>	170 ₄₈	23 ₁₁
	<i>LSTM-ANS</i>	221 ₄₇	32 ₄
	<i>LSTM-APS</i>	237 ₃₅	31 ₄
	<i>LSTM-Ext</i>	197 ₇₉	37 ₂₀
	<i>GGNN-KOT</i>	303 ₁₁₃	28 ₁₀
	<i>GGNN-KOTI</i>	218 ₆₂	20 ₆
	<i>GGNN-Enc</i>	587 ₁₈₂	54 ₁₇
ICST-Rand	<i>LSTM-Raw</i>	62 ₁	303 ₁
	<i>LSTM-ANS</i>	64 ₁	303 ₁
	<i>LSTM-APS</i>	63 ₁	303 ₁
	<i>LSTM-Ext</i>	50 ₀	304 ₂
	<i>GGNN-KOT</i>	325 ₆	301 ₀
	<i>GGNN-KOTI</i>	325 ₆	300 ₀
	<i>GGNN-Enc</i>	326 ₄	300 ₀
ICST-PW	<i>LSTM-Raw</i>	63 ₂	301 ₂
	<i>LSTM-ANS</i>	65 ₂	301 ₆
	<i>LSTM-APS</i>	65 ₂	302 ₂
	<i>LSTM-Ext</i>	52 ₂	303 ₂
	<i>GGNN-KOT</i>	284 ₅₄	250 ₄₇
	<i>GGNN-KOTI</i>	215 ₂₁	194 ₁₇
	<i>GGNN-Enc</i>	245 ₅₀	211 ₅₈
CBMC	<i>LSTM-Raw</i>	115 ₅	8045 ₈₅₇
	<i>LSTM-ANS</i>	100 ₈	7491 ₄₁₄
	<i>LSTM-APS</i>	110 ₄	7273 ₂₅₆
	<i>LSTM-Ext</i>	95 ₆	7333 ₂₉₇
	<i>GGNN-NT</i>	75 ₁₂	974 ₇₃
	<i>GGNN-EncT</i>	62 ₄	834 ₂₁
	<i>GGNN-T</i>	64 ₆	856 ₄₅
JBMC	<i>LSTM-Raw</i>	91 ₁₈	233 ₇₁
	<i>LSTM-ANS</i>	93 ₁₄	181 ₁₂
	<i>LSTM-APS</i>	97 ₁₁	171 ₂₃
	<i>LSTM-Ext</i>	94 ₁₆	287 ₃₁
	<i>GGNN-NT</i>	545 ₂₂₁	25 ₆
	<i>GGNN-EncT</i>	201 ₄₅	12 ₃
	<i>GGNN-T</i>	212 ₇₆	14 ₂

conjecture that the overall accuracy of the other three algorithms dropped because of the larger programs and vocabulary in this dataset. Table 7 shows the dictionary sizes for the LSTM approaches, and Table 8 shows the length of samples for the LSTM approaches (the normal font is the maximum while the smaller font is the mean). Dictionary size refers to the entire vocabulary of unique tokens present in the dataset, whereas length of samples refers

Table 7 Dictionary sizes for the LSTM approaches

Approach	Dictionary size			
	OWASP	ICST	CBMC	JBMC
<i>LSTM-Raw</i>	333	13 237	13 503	942
<i>LSTM-ANS</i>	284	9 724	8 420	842
<i>LSTM-APS</i>	284	9 666	4 053	384
<i>LSTM-Ext</i>	251	4 730	3 404	381

to the length of the token sequence created from the routines mentioned in Section 5.3.2. 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 1 880 average to 16 479 maximum nodes, and 6 411 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 as they are not 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 6). This is mainly due to the large number of parameters being optimized in LSTM and GGNN. However for most of the dataset 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 datasets 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

Table 8 Sample lengths for the LSTM approaches. Numbers in the normal font are the maximum and in the smaller font are the mean

Approach	Sample length			
	OWASP	ICST	CBMC	JBMC
<i>LSTM-Raw</i>	735 224	156393 18524	90923 10134	2425 533
<i>LSTM-ANS</i>	706 212	149886 18104	90923 10133	2425 532
<i>LSTM-APS</i>	706 212	150755 18378	90923 10133	2425 532
<i>LSTM-Ext</i>	925 277	190950 23031	91009 10637	2473 539

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 5) 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 the current 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.

7.2 RQ2: Effect of Data Preparation

Next, we analyze the effect of different data preparation techniques for the ML approaches. *We found LSTM-Ext produced the overall best accuracy results 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.*

In LSTM, we have introduced four code transformation routines. *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 dataset 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, LSTM-Raw has the best results but for the second application scenario LSTM-APS performs better on average.

We also experimented with three node representation techniques for GGNN as mentioned in Section 5.3.3. 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 for the ICST benchmarks are not clear partly because of the distribution of data such as sample lengths, dictionary and dataset sizes (Tables 2, 7 and 8). Moreover, the information encoded in the *GGNN-KOT* and *GGNN-KOTI* approaches is very limited whereas it might be too condensed (aggregated) in *GGNN-Enc* (taking the average over the embeddings of all tokens that appear in the statement), making the training data 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 overall improved 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. For the JBMC dataset, we observe a high test accuracy of 80.15% for GGNN-NT-5 as compared to the 73.95% and 73.00% test accuracy for GGNN-EncT and GGNN-T respectively. 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) might indicate that word2vec embeddings provide some but not much useful information for classification, as the only difference between GGNN-EncT and GGNN-T techniques are the additional embeddings in GGNN-EncT. 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.

As seen in Table 5, *BoW-Occ* and *BoW-Freq* had similar accuracy in general. The largest difference in the test accuracy is 85.53% and 87.14% 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.

7.3 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 5. *Overall, variance was quite different between datasets, and even between different preparations of the same dataset, indicating the different ability of each approach to consistently capture the underlying trend of the datasets.* 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 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 the metrics for the JBMC dataset. The overall best two 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.

We also compare variations in difference between training and test accuracy for the different datasets. For the OWASP we observe little difference between training and test data,

thereby indicating that all the approaches are able to generalize to some extent for this dataset.

For the ICST-Rand dataset the difference between training and test accuracy varies relatively more than the OWASP dataset. We observe that the two approaches with the highest variation in accuracy are HEF and BoW. We hypothesize that this could be due to lack of deep structural and sequence information.

We observe the biggest difference between training and test data among all the benchmarks for the ICST-PW dataset. We believe this could be due to the application scenario, since we the program-wise split could cause algorithms to overfit on the training data (i.e., similar programs with different labels would be always classified as the program in training data, as the model parameters have been tuned for that).

For the CBMC and the JBMC datasets, we observe that the difference for the HEF and BoW approach is much larger than that for the neural network approaches. This variability is similar to the one we observed for the ICST-Rand dataset and reaffirms our hypothesis that it could be due to the lack of structural and sequence information.

7.4 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. Overall, we find that despite achieving lower accuracy, *HEF-J48* and *BoW-Freq* still correctly classified reports that *GGNN* and *LSTM* were unable to, indicating that an ensemble approach (late fusion algorithm) may be beneficial to maximize accuracy.

Figure 9 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 Fig. 9a, 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 Fig. 9b 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 Fig. 9c 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 Fig. 9d 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 (52.1%) were misclassified by at least one approach. The JBMC results in Fig. 9e 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 *LSTM-Ext* and *GGNN-NT* approaches are misclassified by either *HEF-J48* and *BoW-Freq*.

Last, the overall results in Fig. 9 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.

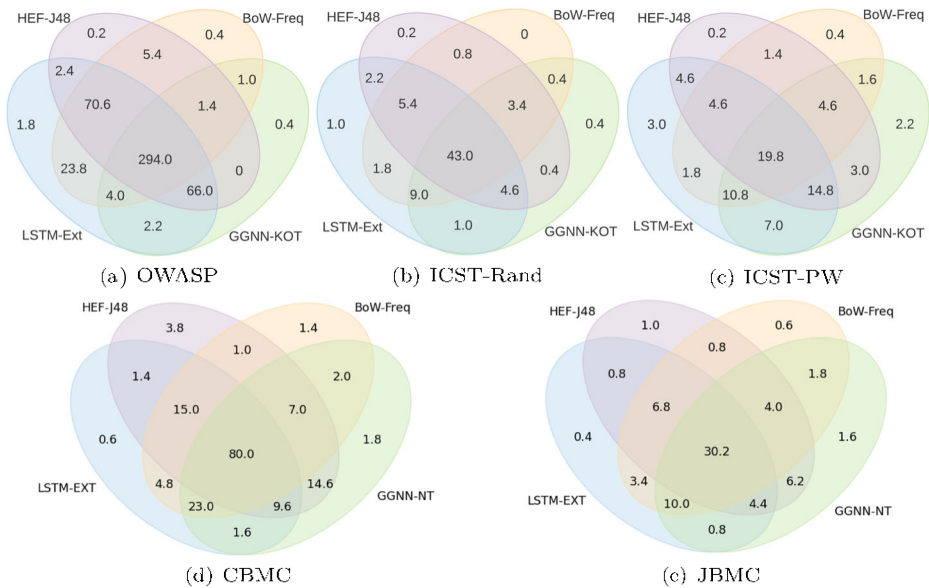


Fig. 9 Venn diagrams of the number of correctly classified examples for *HEF-J48*, *BoW-Freq*, *LSTM-Ext*, and *GGNN-KOT/NT* approaches, average for 5 models trained for the OWASP (a), ICST-Rand (b), ICST-PW (c), CBMC (d), and JBMC (e) datasets (474, 74, 80, 168, and 74 test samples respectively)

Figure 10a shows a sample program from the OWASP dataset to demonstrate the potential advantage of the *LSTM-Ext*. On 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 10b shows the sequential representation used by *LSTM-Ext*. *HEF-J48* used the following feature vector:

```
[rule_name : SQL_INJECTION,
 sink_line : 19, sink_identifier : Statement.executeUpdate,
 source_line : 2, source_identifier : request.getQueryString,
 functions : 4, witness_length : 2, number_bugs : 1, conditions : 1,
 severity : 5, confidence : High, time : 2, classes_involved : 1]
```

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

```

1 public void doPost(HttpServletRequest request, HttpServletResponse response){
2     String param = request.getQueryString();
3     String sql, bar, guess = "ABC";
4     char switchTarget = guess.charAt(1); // 'B'
5     // Assigns param to bar on conditions 'A' or 'C'
6     switch (switchTarget) {
7         case 'A':
8             bar = param; break;
9         case 'B': // always holds
10             bar = "bob"; break;
11         case 'C':
12             bar = param; break;
13         default:
14             bar = "bob's your uncle"; break;
15     }
16     sql = "UPDATE USERS SET PASSWORD='" + bar + "' WHERE USERNAME='foo'";
17     try {
18         java.sql.Statement statement = DatabaseHelper.getSqlStatement();
19         int count = statement.executeUpdate(sql);
20     } catch (java.sql.SQLException e) {
21         throw new ServletException(e);
22     }

```

(a) A program from the OWASP dataset.

```

1 org owasp benchmark UNK UNK do Post ( Http Servlet Request Http Servlet
2 Response ) : String VAR 6 = p 1 request get Query String ( ) : C VAR 10 = STR 1
3 char At ( 1 ) : switch VAR 10 : String Builder VAR 14 = new String Builder :
4 String Builder VAR 18 = VAR 14 append ( STR 0 ) : String Builder VAR 20 = VAR
5 18 append ( VAR 13 ) : String Builder VAR 23 = VAR 20 append ( STR 3 ) : String
6 VAR 25 = VAR 23 to String ( ) : java sql Statement VAR 27 = get Sql Statement
7 ( ) : I VAR 29 = VAR 27 execute Update ( VAR 25 ) :
8 PHI VAR 13 = VAR 6 STR 4 VAR 6 STR 2

```

(b) Corresponding sequential representation used for *LSTM-Ext*.

Fig. 10 An example program (simplified) from the OWASP dataset that was correctly classified only by *LSTM-Ext* and the sequential representation used for *LSTM-Ext*

on correlations that might exist for the features above. For this example, such correlations weigh more towards the true positive decision, thus lead to a misclassification.

On the other hand, the *LSTM-Ext* representation includes the program information (Fig. 10b). 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` on line 3 corresponding to the switch statement. Then, we see string and SQL operations through lines 4 to 7, followed by a PHI instruction on 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 Fig. 10b without their order. This suggests that the overall correlation of the tokens that appear in this slice does not favor the false positive class. We believe that the correct classification by *LSTM-Ext* was thus not due to the presence of certain tokens, but rather due to the sequential structure.

Figure 11a shows a sample program from the JBMC dataset to demonstrate another potential advantage of the *LSTM-Ext* and *GGN*. On line 6, the variable `int j` can cause an `ArithmeticException` when the denominator variable `denom` is set to 0. However the JBMC tool fails to capture this violation and marks it safe, thereby creating a false positive

```

1 import org.sosy_lab.sv_benchmarks.Verifier;
2 public class Main {
3     public static void main(String[] args) {
4         int denom = Verifier.nondetInt();
5         try {
6             int j = 10 / denom;
7         } catch (ArithmeticException exc) {
8             assert false;
9         }
10    }
11 }

```

(a) A program from the JBMC dataset.

```

1 import org . sosy lab . sv benchmarks . Verifier ;
2 public class Main {
3     public static void main ( String [ ] args ) {
4         int VAR1 = Verifier . nondetInt ( ) ;
5         try {
6             int VAR2 = N2 / VAR1 ;
7         } catch ( UNK exc ) {
8             assert false ;
9         }
10    }
11 }

```

(b) Corresponding sequential representation used for *LSTM-Ext*.

```

1 numIfs: 0, numVars: 10, numFuncs: 2, numCalls: 2, numLoops: 0, numArrayTypes:
  0, numCompositeTypes: 1, numFundamentalTypes: 9, numLines: 15,
  SSASwitchInstruction: 1, SSAArrayLengthInstruction: 0,
  SSAArrayLoadInstruction: 0, SSAPutInstruction: 0,
  SSAComparisonInstruction: 2, SSAConditionalBranchInstruction: 1,
  SSANewInstruction: 0, SSAReturnInstruction: 0, SSAGetInstruction: 0,
  SSAGotoInstruction: 3, SSALoadMetadataInstruction: 1,
  SSAInstanceofInstruction: 1, SSAArrayStoreInstruction: 1,
  SSACheckCastInstruction: 2, SSAGetCaughtExceptionInstruction: 0,
  SSABinaryOpInstruction: 2, SSAPhiInstruction: 0, SSAUnaryOpInstruction:
  1, SSAConversionInstruction: 0, SSAPiInstruction: 0,
  SSAMonitorInstruction: 0, SSAInvokeInstruction: 1, SSAThrowInstruction: 1

```

(c) Corresponding sequential representation used for *HEF*.

```

1 "graph": [[0, 0, 1], [1, 0, 2], [1, 0, 3], [3, 0, 4], [3, 0, 5], [3, 0, 6],
  [1, 0, 7], [7, 0, 8], [8, 0, 9], [8, 0, 10], [7, 0, 11], [7, 0, 12], [7,
  0, 13], [7, 0, 14], [1, 0, 15]],
2 "node_features": [[9.0, -1.0, -1.0], [39.0, -1.0, -1.0], [32.0, 16.0, -1.0],
  [14.0, -1.0, -1.0], [42.0, 17.0, 10.0], [29.0, 213.0, -1.0], [23.0,
  217.0, 220.0], [21.0, -1.0, -1.0], [34.0, -1.0, -1.0], [2.0, 115.0,
  203.0], [32.0, 126.0, -1.0], [32.0, 137.0, -1.0], [4.0, 224.0, 225.0],
  [29.0, 223.0, -1.0], [23.0, 217.0, 220.0], [32.0, 148.0, -1.0]]

```

(d) Corresponding sequential representation used for *GGN-2T*.

Fig. 11 An example program (simplified) from the JBMC dataset that was correctly classified by *LSTM-Ext* and *GGN-2T*

sample. Since the *HEF* approach depends on the specific summarized code features and information that can be obtained from the static analysis report, it cannot detect the violation as it requires contextual information. However as the *LSTM-Ext* has the entire sequence

with contextual information as seen in Fig. 11b, it could detect the violation due to tokens that denote that an integer $N2$, is divided by another integer *denom*. As *LSTM* can retain information over long sequences it can make use of this information to detect the violation. Similarly for the *GGN-2T* approach the violation can be detected using the relationship formed by the edges of the nodes denoting the variable and operation in the AST.

7.5 Threats to Validity

We identify various internal and external threats to the validity of our study.

Internal Threats One internal threat to validity is that we did not have any datasets on which we compared the AST- and PDG-based representations. We do not view this threat as major because the goal of this work was not to compare the two representations, but rather, explore their viability as program representations. Additionally, we do not claim that the representations we propose are the best for the given datasets.

Second, since we apply data preparation routines cumulatively, it is possible that improved performance may be caused by the interaction between multiple routines that we do not account for. Note that we do not claim that the steps we propose are optimal, just that they are viable for training high-performance machine learning models.

Finally, it is possible that there are some confounding variables we are not aware of that could impact our results. Kang, Aw, and Lo pointed out that in many studies, the ground truth leaks into the test data through feature extraction (2022). We did not use any of the features they identified as responsible for data leakage (i.e., warning context and defect likelihood), and we further attempted to mitigate this threat by post-processing to remove things like variable names that might give away the ground truths.

External Threats First, the datasets may not be representative. Indeed, the OWASP, CBMC, and JBMC program sets are synthetic. Therefore, we collected the first real-world (ICST) dataset for classifying SA results, consisting of 14 programs to increase the generalizability of our results. Using a larger dataset would be challenging, since few datasets for static analysis exist with known ground truths and manually classifying results is time-consuming. We believe the mix of programs we chose, which includes programs of various sizes written by various people, helps mitigate this threat.

Second, the configurations we chose specifically for CBMC and JBMC may not be representative of real configurations. In order to train our models, a balanced dataset (i.e., a dataset with significant amounts of both correct and incorrect results) was necessary. We obtained configurations that gave a balanced dataset from previous work that used covering arrays (Koc et al. 2021), and it is unlikely that a user would use these exact configurations. This threatens the generality of our approach to real scenarios if the configurations we selected do not resemble a typical user experience. It is well known that false positives are a major barrier to the usage of analysis tools (Johnson et al. 2013), so we do not believe our scenario and the requirement of a balanced dataset are unrealistic. One potential solution we chose not to adapt would be rebalancing the dataset. This would consist of using the default configuration, rebalancing the training data via oversampling, and testing on the original distribution. However, this approach would introduce an additional significant threat. Recall that the default configurations of CBMC and JBMC are tuned to be sound and precise, but often fail to terminate. If we kept the classifications from the default configuration, and discarded non-terminations (since we only classify correct/incorrect and not

terminated/non-terminated), all of the results in the test set would be labeled “correct” and a naive classifier that classified everything as correct would achieve 100% accuracy.

Third, FindSecBugs, CBMC, and JBMC may not be representative of real-world static analyzers. FindSecBugs performs information flow analysis and is specialized to find security bugs, and CBMC and JBMC are both model checkers. There are various other forms of static analysis that we do not study, and our results may not generalize to those. We tried to mitigate this threat by ensuring the set of tools we used implemented different analysis techniques and analyzed different languages.

Fourth, our ICST dataset consists of 400 data points which may not be large enough to train neural networks with high confidence. We repeat the experiments using different random seeds and data splits to analyze the variability that might be caused by having limited data. However we still suspect that the models overfit due to the big difference between train and test accuracy. However, none of the overall observations we make rely solely on trends observed from the ICST benchmarks.

Finally, we ran our experiments between 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.

8 Discussion

We now summarize our key insights and their implications for both researchers and practitioners. The key implications of this work for researchers are as follows:

- *More research on representing programs for learning is needed (RQ1,RQ2).* In this work, we experimented with various representations for programs, and we saw that different representations provided different tradeoffs. For example, the graph representation of ASTs outperformed the vector representation (i.e., GGNN outperformed LSTM on CBMC and JBMC datasets). However, the same was not true for programs whose graph representation was a slice—LSTM outperformed GGNN on these programs, despite the slice being designed to only contain useful information. More research is warranted to understand the tradeoffs between these models.
- *Different approaches provide different tradeoffs that need to be better understood (RQ4).* As shown in Fig. 9, in no circumstance did LSTM and GGNN completely subsume traditional approaches. This indicates the potential utility of an ensemble approach that combines neural networks and traditional models, and it also motivates further research into the tradeoffs different models present.

The primary implication of this work for practitioners is:

- *While ML-based filtering approaches may not be sound, they can increase productivity with regard to the number of bugs fixed in a certain amount of time.* In scenarios wherein finding every bug is critical, ML approaches are not a good choice, as they can filter out true results in addition to false results. However, if a practitioner’s goal is to fix as many bugs as they can in a certain amount of time, ML classifiers can help optimize this time by reducing the effective false positive rate. Taking the ICST dataset as an example, which contained approximately the same number of true positives and false positives, the maximum accuracy we were able to achieve was 89.33%. This means that we can expect the model to misclassify about 43 reports out of the 400 total. Since the precision and recall values were about the same, we can assume about half of these

will be false positives misclassified as true, and the other half true bugs misclassified as false. If the practitioner only manually classifies the results judged as true positives, they would save half the total investigation time at the cost of approximately 21 missed bugs.

9 Related Work

There are several threads of related work.

ICST 2019 The current paper extends our prior work (Koc et al. 2019) with two major additions: more datasets and additional and improved ML approaches. Compared to our previous work, which used the OWASP and ICST program set, we added the CBMC and JBMC program set from the annual SV-COMP (Beyer 2018, 2019), as well as the ground truth dataset obtained by running CBMC and JBMC on them. This allowed us to improve our previous study and make more general findings by evaluating two different types of static analysis. Further details about the datasets are presented in Section 4. We also significantly extended our approach to learning. This work added ASTs for representing programs from the CBMC and JBMC datasets in addition to the program slices used in the previous work (details in Section 5.2). As such, we had to adapt all of our data preparation routines to accommodate the new data. Specifically, we defined new features to be extracted for HEF representations from the CBMC and JBMC datasets (details in Section 5.3.1). We also adapted the transformations used to transform program representations into sequences of tokens for the ASTs created from the CBMC and JBMC datasets (details in Section 5.3.2). We also extended the GGNN implementation from Microsoft Research (2019) to use it with ASTs created from the CBMC and JBMC datasets, and added new node representations for these datasets to be used in the GGNN implementation (details in Section 5.3.3). Our findings from the extended work were broadly the same as in the previous work, with more evidence backing them. We still find that neural networks outperform but do not subsume traditional approaches, and that data preparation is critical and warrants further study. An additional extension of our work in this paper is the additional dimension of true and false negatives in the CBMC and JBMC datasets. In our original work, the models only had to differentiate between true and false positive results. Now, our correct class includes both true negatives and positives, and our incorrect class includes both false negatives and positives. Our work in this paper shows that ML models are still capable of this classification task.

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; Yüksel and Sözer 2013; Tripp et al. 2014; Utture et al. 2022; Kang et al. 2022). For instance, Tripp et al. (2014) identify 14 such features for false positive cross-site scripting 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. Utture et al. (2022) used random forests to prune imprecise call graphs. Their models are trained on features extracted from the static call graph, while labels were obtained from the dynamic call graph. Their approach was able to improve the precision of call graphs produced by WALA (IBM 2006), DOOP (Bravenboer and Yannis 2009), and Petablox (Naik 2020). Their approach focuses on achieving balance in a call graph, which, in practice, means trying to prune mostly false positive edges but allowing for true positive edges to

be pruned as well. Our approach and models are designed to evaluate the ability of models to prune only false positives while retaining true positives. Furthermore, in addition to features from the bug reports themselves, our hand-engineered features include things extracted from the source code. 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. Zhou et al. (2019) developed a general graph neural network based model for graph-level classification through learning on a rich set of code semantic representations which they tested on four large open-source C projects: the Linux Kernel, QEMU, Wireshark, and FFmpeg. Tanwar et al. (2020) developed a novel AI-based system that uses ASTs created from the source code and an active feedback loop to identify and bugs during source code development. They tested their approach on the Cisco codebase for C and C++ programming language. Kharkar et al. (2022) developed a transformer-based learning approach to identify false positive bug warnings, and they tested their approach on a custom dataset consisting of warnings from Infer for various open source and proprietary software projects. Raghothaman et al. (2018) build on an ML approach, Bayesian inference, that additionally relies on direct feedback from human tool users. These works all focus on a single learning approach on analyses targeting a single language, while our work uniquely evaluates many different models on analyses for two languages. Furthermore, none of the existing work in this line of research has studied the second application scenario we identify for ML-assisted triage, which tries to generalize learning to new programs.

NLP Techniques Applied to Code Multiple researchers have successfully applied NLP techniques to programs to tackle software engineering problems such as clone detection (White et al. 2016), API mining (Gu et al. 2016; Fowkes and Sutton 2016), variable naming and renaming (Raychev et al. 2015; Allamanis et al. 2015), code suggestion and completion (Tu et al. 2014; Nguyen et al. 2013; Raychev et al. 2014), and bug detection (Allamanis et al. 2017). Allamanis et al. (2018) conducted an extensive survey of such research efforts. There has also been recent work 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, and some datasets for the same problem have been published (Wang et al. 2020; Li et al. 2019; Wan et al. 2019). There has also been research to develop unsupervised embeddings for software libraries, which can then be used for any of the above mentioned tasks (Feng et al. 2020; Chen and Monperrus 2019; Alon et al. 2019). However, none of these efforts addresses the problem of identifying and distinguishing incorrect SA reports.

10 Conclusions and Future Work

We presented an empirical study that evaluates three families of ML approaches—traditional approaches, recurrent neural networks, and graph neural networks—to classifying static analysis results from three tools—FindSecBugs, CBMC, and JBMC—as either correct or incorrect. To adapt these approaches to this classification task, we introduced new code transformations for preparing code as input to each ML approach. We used

multiple datasets for evaluation: the OWASP and ICST datasets, consisting of classified FindSecBugs reports; and the CBMC and JBMC datasets, consisting of verification results. We also studied two application scenarios: one in which the approach is being used continually as software is developed, thereby having similar programs in the test and training sets; and one in which the user applies the approach to new software and so has different programs in the test and training sets.

The results of our experiments suggest that neural network approaches work better than traditional 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 potential usage scenarios, and found that the application scenario in which the training set and test set contain different programs is more challenging. This was shown by the lower accuracy on the CBMC and JBMC datasets compared to OWASP, as well as the apparent degree of overfitting on the ICST-PW dataset as opposed to the ICST-Rand dataset. However, in this application scenario, we observed that more detailed data preparation with abstraction and word extraction leads to significant increases in accuracy. Finally, we 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 embeddings for either node representation or as input to recurrent neural networks. We also plan to explore the configuration spaces of these tools, potentially integrating configuration into the learning infrastructure, similarly to previous work (Koc et al. 2021).

Acknowledgments 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.

Data Availability There were 5 datasets used during this study. 2 datasets (CBMC and JBMC) were created using the SV-COMP dataset, available at <https://github.com/sosy-lab/sv-benchmarks>. Steps on how to use the SV-COMP dataset, the other 3 datasets (OWASP, ICST-Rand, and ICST-PW) along with the experimental infrastructure of this study are available in the replication repository at <https://bitbucket.org/SaiArrow/emse-replication-package/>.

Declarations

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.

References

- Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y, Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, Mané D, Monga R, Moore S, Murray D, Olah C, Schuster M, Shlens J, Steiner B, Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V, Viégas F, Vinyals O, Warden P, Wattenberg M, Wicke M, Yu Y, Zheng X (2015) Tensorflow: large-scale machine learning on heterogeneous systems. <https://www.tensorflow.org/> Software available from tensorflow.org

- Allamanis M, Barr ET, Bird C, Sutton C (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, pp 38–49, <https://doi.org/10.1145/2786805.2786849>
- Allamanis M, Brockschmidt M, Khademi M (2017) Learning to represent programs with graphs. arXiv:1711.00740 [cs]
- Allamanis M, Barr ET, Devanbu P, Sutton C (2018) A survey of machine learning for big code and naturalness. *ACM Comput Surv* 51(4):Article 81, 37 pp. <https://doi.org/10.1145/3212695>
- Alon U, Zilberstein M, Levy O, Yahav E (2019) code2vec: learning distributed representations of code. In: Proceedings of the ACM on programming languages 3, POPL, pp 1–29
- Andres M (2013) Free chat-server: a chatserver written in Java. <https://sourceforge.net/projects/freecs>
- Apollo 2018 (2018) Apollo: a distributed configuration center. <https://github.com/ctripcorp/apollo>
- Arteau Ph, Formáánek D, Polešovský T (2018) Find security bugs, version 1.4.6. <http://find-sec-bugs.github.io>, Accessed: 2022-07-19
- AutoML (2022) AutoML. <https://www.automl.org/automl>
- Beyer D (2018) Results of the competition. <https://sv-comp.sosymlab.org/2018/results/results-verified/>. Accessed: 2021-04-22
- Beyer D (2019) Results of the competition. <https://sv-comp.sosymlab.org/2019/results/results-verified/>. Accessed: 2021-04-22
- Biere A, Cimatti A, Clarke E, Zhu Y (1999) Symbolic model checking without BDDs. In: Cleaveland WR (ed) Tools and algorithms for the construction and analysis of systems. Springer, Berlin, pp 193–207
- Blackburn SM, Garner R, Hoffmann C, Khang AM, McKinley KS, Bentzur R, Diwan A, Feinberg D, Frampton D, Guyer SZ, Hirzel M, Hosking A, Jump M, Lee H, Moss JEB, Phansalkar A, Stefanovic D, VanDrunen T, von Dincklage D, Wiedermann B (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, pp 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>
- Bravenboer M, Yannis S (2009) Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not* 44(10):243–262. <https://doi.org/10.1145/1639949.1640108>
- Burato E, Ferrara P, Spoto F (2017) Security analysis of the OWASP benchmark with julia. In: Proceedings of ITASEC17, the 1st Italian conference on security, Venice, Italy
- Carrier P-L, Cho K (2018) LSTM networks for sentiment analysis: deeplearning 0.1 documentation. <http://deeplearning.net/tutorial/lstm.html>
- Chen Z, Monperrus M (2019) A literature study of embeddings on source code. arXiv:1904.03061
- Cho K, van Merriënboer B, Bahdanau D, Bengio Y (2014) On the properties of neural machine translation: encoder-decoder approaches. <https://doi.org/10.48550/ARXIV.1409.1259>
- Clarke E, Kroening D, Lerda F (2004) A tool for checking ANSIC programs. In: Jensen K, Podolski A (eds) Tools and algorithms for the construction and analysis of systems (TACAS 2004) (Lecture Notes in Computer Science), vol 2988. Springer, pp 168–176
- Cordeiro L, Kesseli P, Kroening D, Schrammel P, Marek T (2018) JBMC: a bounded model checking tool for verifying java bytecode. In: Computer aided verification (CAV) (LNCS), vol 10981. Springer International Publishing, Cham, pp 183–190
- Dam HK, Tran T, Pham TTM (2016) A deep language model for software code. In: FSE 2016: proceedings of the foundations software engineering international symposium, pp 1–4
- Diamantopoulos T (2020) ASTEXtractor v0.5. <https://github.com/thdianman/ASTExtractor>
- Eclipse Foundation (2022a) Eclipse java integrated development environment. <https://www.eclipse.org/ide/>
- Eclipse Foundation (2022b) Jetty: lightweight highly scalable java based web server and servlet engine. <https://www.eclipse.org/jetty>
- Eibe F, Hall MA, Witten IH (2016) The WEKA workbench. Morgan Kaufmann
- Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D, Zhou M (2020) CodeBERT: a pre-trained model for programming and natural languages. arXiv:2002.08155
- Ferrante J, Ottenstein KJ, Warren JD (1987) The program dependence graph and its use in optimization. *ACM Trans Program Lang Syst* 9(3):319–349. <https://doi.org/10.1145/24039.24041>
- Fowkes J, Sutton C (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, pp 254–265, <https://doi.org/10.1145/2950290.2950319>
- Gers FA, Schmidhuber J, Fred C (2000) Learning to forget: continual prediction with LSTM. *Neural Comput* 12(10):2451–2471
- Giraph (2020) Giraph: large-scale graph processing on Hadoop. <http://giraph.apache.org>

- Goldberg Y (2017) Neural network methods for natural language processing. *Synth Lect Hum Lang Technol* 10(1):1–309
- Goldberg Y, Levy O (2014) word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method. arXiv:1402.3722
- Gori M, Monfardini G, Scarselli F (2005) A new model for learning in graph domains. In: 2005 IEEE International joint conference on neural networks, 2005. IJCNN'05. Proceedings, vol 2. IEEE, pp 729–734
- Gros D, Sezhiyan H, Devanbu P, Yu Z (2020) Code to comment “translation”: data, metrics, baselining & evaluation. arXiv:2010.01410
- Gu X, Zhang H, Zhang D, Kim S (2016) Deep API learning. In: Proceedings of the 2016 24th ACM SIGSOFT International symposium on foundations of software engineering. ACM, pp 631–642
- h2db (2022) H2 database engine. <http://www.h2database.com>
- Haque S, LeClair A, Wu L, McMillan C (2020) Improved automatic summarization of subroutines via attention to file context. In: Proceedings of the 17th international conference on mining software repositories, <https://doi.org/10.1145/3379597.3387449>
- Heckman SS (2007) Adaptive probabilistic model for ranking code-based static analysis alerts. In: 29th International conference on software engineering—companion. ICSE 2007 companion, pp 89–90, <https://doi.org/10.1109/ICSECOMPANION.2007.16>
- Heckman SS (2009) A systematic model building process for predicting actionable static analysis alerts. North Carolina State University
- Hochreiter S, Schmidhuber J (1997) Long short-term memory. *Neural Comput* 9(8):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>
- Johnson A, Wayne L, Moore S, Chong S (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, pp 291–302, <https://doi.org/10.1145/2737924.2737957>
- Johnson B, Song Y, Murphy-Hill E, Bowdidge R (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, pp 672–681. <http://dl.acm.org/citation.cfm?id=2486788.2486877>
- Jozefowicz R, Zaremba W, Sutskever I (2015) An empirical exploration of recurrent network architectures. In: Proceedings of the 32nd international conference on international conference on machine learning—volume 37 (ICML'15). JMLR.org, pp 2342–2350
- Kang HJ, Aw KL, Lo D (2022) Detecting false alarms from automatic static analysis tools: how far are we? In: Proceedings of the 44th international conference on software engineering (ICSE '22). Association for Computing Machinery, New York, pp 698–709, <https://doi.org/10.1145/3510003.3510214>
- Kharkar A, Moghaddam RZ, Jin M, Liu X, Shi X, Clement C, Sundaresan N (2022) Learning to reduce false positives in analytic bug detectors. In: Proceedings of the 44th international conference on software engineering. ACM, <https://doi.org/10.1145/3510003.3510153>
- Kingma DP, Adam JB (2014) A method for stochastic optimization. <https://doi.org/10.48550/ARXIV.1412.6980>
- Koc U, Saadatpanah P, Foster JS, Porter AA (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, pp 35–42, <https://doi.org/10.1145/3088525.3088675>
- Koc U, Wei S, Foster JS, Carpuat M, Porter AA (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), pp 288–299, <https://doi.org/10.1109/ICST.2019.00036>
- Koc U, Mordahl A, Wei S, Foster JS, Porter A (2021) SATune: 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
- Kroening D, Tautschnig M (2014) CBMC—C bounded model checker. In: Ábrahám E, Havelund K (eds) Tools and algorithms for the construction and analysis of systems. Springer, Berlin, pp 389–391
- Kushman N, Barzilay R (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, pp 826–836
- Li H, Kim S, Chandra S (2019) Neural code search evaluation dataset. arXiv:1908.09804
- Li Y, Tarlow D, Brockschmidt M, Zemel R (2015a) Gated graph sequence neural networks. <https://doi.org/10.48550/ARXIV.1511.05493>

- Li Y, Tarlow D, Brockschmidt M, Zemel R (2015b) Gated graph sequence neural networks. arXiv:1511.05493
- Ling W, Blunsom P, Grefenstette E, Hermann KM, Kociský T, Wang F, Senior A (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, pp 599–609
- LLVM Team (2020) The LLVM compiler infrastructure. <https://github.com/llvm/llvm-project.git>
- Lukins SK, Kraft NA, Letha HE (2010) Bug Localization using latent Dirichlet allocation. *Inf Softw Technol* 52(9):972–990. <https://doi.org/10.1016/j.infsof.2010.04.002>
- Mandic DP, Chambers J (2001) Recurrent neural networks for prediction: learning algorithms architectures and stability. Wiley, New York
- Microsoft (2019) Microsoft gated graph neural networks. <https://github.com/Microsoft/gated-graph-neural-network-samples>
- Mikolov T, Chen K, Corrado G, Dean J, Sutskever L, Zweig G (2013) word2vec. <https://code.google.com/p/word2vec>
- Mohr M, Hecker M, Bischof S, Bechberger J (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>
- Naik M (2020) Petablox: large-scale software analysis and analytics using datalog. Technical Report. Georgia Tech Research Inst Atlanta Atlanta United States
- NASA Ames Research Center (2022) Java pathfinder. <https://github.com/javapathfinder>
- Nguyen TT, Nguyen SE, Nguyen HA, Nguyen TN (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, pp 532–542. <https://doi.org/10.1145/2491411.2491458>
- Nie C, Hareton L (2011) A survey of combinatorial testing. *ACM Comput Surv* 43(2):Article 11, 29 pp. <https://doi.org/10.1145/1883612.1883618>
- OWASP (2014) The OWASP Benchmark for Security Automation, version 1.1. <https://www.owasp.org/index.php/Benchmark>. Accessed: 2018-01-04
- Panthapilackel S, Nie P, Gligoric M, Li JJ, Mooney RJ (2020) Learning to update natural language comments based on code changes. arXiv:2004.12169
- Prlić A, Yates A, Bliven SE, Rose PW, Jacobsen J, Troshin PV, Chapman M, Gao J, Koh CH, Foisy S et al (2012) Biojava: an open-source framework for bioinformatics in 2012. *Bioinformatics* 28(20):2693–2695
- Quinlan JR (2014) C4.5: programs for machine learning. Elsevier
- Raghothaman M, Kulkarni S, Heo K, Naik M (2018) User-guided program reasoning using bayesian inference. In: Proceedings of the 39th ACM SIGPLAN conference on programming language design and implementation (PLDI 2018). ACM, New York, pp 722–735. <https://doi.org/10.1145/3192366.3192417>
- Raychev V, Vechev M, Yahav E (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, pp 419–428. <https://doi.org/10.1145/2594291.2594321>
- Raychev V, Vechev M, Krause A (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, pp 111–124. <https://doi.org/10.1145/2676726.2677009>
- Rish I et al (2001) An empirical study of the naive Bayes classifier. In: IJCAI 2001 Workshop on empirical methods in artificial intelligence, vol 3, pp 41–46
- Rosen BK, Wegman MN, Zadeck FK (1988) Global value numbers and redundant computations. In: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL '88). ACM, New York, pp 12–27. <https://doi.org/10.1145/73560.73562>
- Rosenblatt F (1958) The perceptron: a probabilistic model for information storage and organization in the brain. *Psychol Rev* 65(6):386
- Russell SJ, Norvig P (2016) Artificial intelligence: a modern approach. Pearson Education Limited, Malaysia
- Safavian SR, Landgrebe D (1991) A survey of decision tree classifier methodology. *IEEE Trans Syst Man Cybern* 21(3):660–674. <https://doi.org/10.1109/21.97458>
- Sak H, Senior A, Beaufays F (2014) Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In: Fifteenth annual conference of the international speech communication association
- Scarselli F, Gori M, Tsoi AC, Hagenbuchner M, Monfardini G (2009) The graph neural network model. *IEEE Trans Neural Netw* 20(1):61–80. <https://doi.org/10.1109/TNN.2008.2005605>
- Smith A (2019) Universal password manager. <http://upm.sourceforge.net>
- Sureka A, Jalote P (2010) Detecting duplicate bug report using character N-Gram-Based features. In: 2010 Asia pacific software engineering conference, pp 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>
- Tanwar A, Sundaresan K, Ashwath P, Ganesan P, Chandrasekaran SK, Ravi S (2020) Predicting vulnerability in large codebases with deep code representation. <https://doi.org/10.48550/ARXIV.2004.12783>
- 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.lldm.org/12.0.0/tools/clang/docs/index.html>
- The HSQL Development Group (2021) HyperSQL DataBase. <http://hsqldb.org>
- Theano Development Team (2016) Theano: a python framework for fast computation of mathematical expressions. arXiv:1605.02688
- Thunes C (2020) javalang: pure Python Java parser and tools. <https://pypi.org/project/javalang/>. Accessed: 2022-02-13
- Tripp O, Guarnieri S, Pistoia M, Aleksandr A (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, pp 762–774, <https://doi.org/10.1145/2660267.2660339>
- Tu Z, Su Z, Devanbu P (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, pp 269–280, <https://doi.org/10.1145/2635868.2635875>
- Utture A, Liu S, Kalhauge CG, Palsberg J (2022) Striking a balance: pruning false-positives from static call graphs. In: Proceedings of the 44th international conference on software engineering (ICSE '22). Association for Computing Machinery, New York, pp 2043–2055, <https://doi.org/10.1145/3510003.3510166>
- Wan Y, Shu J, Sui Y, Xu G, Zhao Z, Wu J, Yu PS (2019) Multi-modal attention network learning for semantic source code retrieval. arXiv:1909.13516
- Wang J, Wang S, Wang Q (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 (ESEM '18). ACM, New York, p Article 17, 10 pp, <https://doi.org/10.1145/3239235.3239523>
- Wang W, Zhang Y, Zeng Z, Xu G (2020) Trans3: a transformer-based framework for unifying code summarization and code search. arXiv:2003.03238
- Weiser M (1981) Program slicing. In: Proceedings of the 5th international conference on software engineering. IEEE Press, pp 439–449
- White M, Tufano M, Vendome C, Poshvanyk D (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, pp 87–98, <https://doi.org/10.1145/2970276.2970326>
- Xypolytos A, Xu H, Vieira B, Ali-Eldin AMT (2017) A framework for combining and ranking static analysis tool findings based on tool performance statistics. In: 2017 IEEE International conference on software quality, reliability and security companion (QRS-c). IEEE, pp 595–596
- Ye X, Shen H, Ma X, Bunescu R, Liu C (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, pp 404–415, <https://doi.org/10.1145/2884781.2884862>
- Yüksel U, Sözer H (2013) Automated classification of static code analysis alerts: a case study. In: 2013 IEEE International conference on software maintenance, pp 532–535
- Zeiler MD (2012) ADADELTA: an adaptive learning rate method. arXiv:1212.5701
- Zhou Y, Liu S, Siow J, Du X, Liu Y (2019) Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. <https://doi.org/10.48550/ARXIV.1909.03496>

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Affiliations

Sai Yerramreddy¹  · **Austin Mordahl²** · **Ugur Koc¹** · **Shiyi Wei²** · **Jeffrey S. Foster³** · **Marine Carpuat¹** · **Adam A. Porter¹**

Austin Mordahl
austin.mordahl@utdallas.edu

Ugur Koc
ukoc@cs.umd.edu

Shiyi Wei
swei@utdallas.edu

Jeffrey S. Foster
jfoster@cs.tufts.edu

Marine Carpuat
marine@cs.umd.edu

Adam A. Porter
aporter@cs.umd.edu

¹ Department of Computer Science, University of Maryland, College Park, MD, USA

² Department of Computer Science, University of Texas at Dallas, Richardson, TX, USA

³ Department of Computer Science, Tufts University, Medford, MA, USA