Learning a Classifier for False Positive Error Reports Emitted by Static Code Analysis Tools

Ugur Koc Parsa Saadatpanah Jeffrey S. Foster Adam A. Porter
University of Maryland, College Park, USA
{ukoc, parsa, jfoster, aporter}@cs.umd.edu

Abstract

The large scale and high complexity of modern software systems make perfectly precise static code analysis (SCA) infeasible. Therefore SCA tools often over-approximate, so not to miss any real problems. This, however, comes at the expense of raising false alarms, which, in practice, reduces the usability of these tools.

To partially address this problem, we propose a novel learning process whose goal is to discover program structures that cause a given SCA tool to emit false error reports, and then to use this information to predict whether a new error report is likely to be a false positive as well. To do this, we first preprocess code to isolate the locations that are related to the error report. Then, we apply machine learning techniques to the preprocessed code to discover correlations and to learn a classifier.

We evaluated this approach in an initial case study of a widely-used SCA tool for Java. Our results showed that for our dataset we could accurately classify a large majority of false positive error reports. Moreover, we identified some common coding patterns that led to false positive errors. We believe that SCA developers may be able to redesign their methods to address these patterns and reduce false positive error reports.

CCS Concepts • **Theory of computation** → *Program analysis*; • **Computing methodologies** → *Machine learning algorithms*; *Supervised learning by classification*; • **Software and its engineering** → *General programming languages*

Keywords Static code analysis, program slicing, Naive Bayes classifier, long short-term memories

1. Introduction

Static code analysis (SCA) is the process of analyzing a program's source code to find flaws without executing it. SCA tools help developers identify weaknesses and flaws that might jeopardize the security and integrity of a software program. Although SCA tools can clearly aid developers, they are also known to generate large numbers of spurious error reports, i.e., false positives. Simplifying greatly, this happens because SCA tools rely on approximations and assumptions that help their analyses scale to large and complex software systems. The trade-off is that analysis results become imprecise, leading to false positives. As a result, developers often find themselves sifting through the false alarms to find and solve the real flaws. Inevitably, some developers stop inspecting the SCA tool's output.

To solve this problem, some previous research efforts have applied machine learning techniques to filter out the false positive error reports (Kremenek and Engler 2003; Yüksel and Sözer 2013; Tripp et al. 2014). At a high level, these efforts learn classifiers by using a set of features collected from reports and SCA tools. We found, however, that none of these efforts incorporates information about the structure of the actual code being analyzed. We hypothesize that adding detailed knowledge of a program's structure to the machine learning process can lead to more effective classifiers. Therefore, in this work, we propose an approach for learning a classifier by training on the code itself. Our goal is to discover the program structures correlated with false positive error reports and then to filter out such error reports by detecting whether these program structures exist in a program that has reported a potential flaw.

In the proposed approach, the first step involves reducing the code to a smaller form of itself that leads to the error report. In particular, we try two different code reduction techniques. First, as a very naive reduction, we simply take the body of the method that contains the warning line. Second, as a more precise reduction, we compute a backward slice with respect to the warning line. Next, using the reduced code, we discover program structures that are correlated with false positive error reports using an easy to interpret machine learning technique. Finally, also using the reduced code, we learn a classifier to filter out the false positive error reports using more advanced machine learning techniques (see more details in Section 2).

We evaluated the proposed approach by conducting a case study in which we evaluated the two code reduction techniques and two machine learning models to classify $2,371$ error reports produced by FindSecBugs plug-in of FindBugs (Arteau 2016; Ayewah et al. 2008) by training on Java bytecode representation of the programs that lead to the error reports. As a simple learning model, we used Naive Bayes Inference. Then, as a more sophisticated learning model, we used long short-term memories (LSTM) (more details in Section 3).

The results of these experiments are promising. First, in the best case, our approach achieved 97% recall and 85% accuracy. In addition, our analysis of the resulting learning models revealed likely causes explaining why the SCA tool produced certain false positive reports (see Section 4).

2. Approach

Figure 1 depicts the proposed learning approach. Given source code and a list of error reports emitted by an SCA tool for the source code, we start by reducing the source code to a subset of itself to isolate the code locations that are related to the error report. Next, we label the error reports by manually examining code (potentially

MAPL'17, June 18, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-5071-6/17/06...$15.00
http://dx.doi.org/10.1145/3088525.3088675

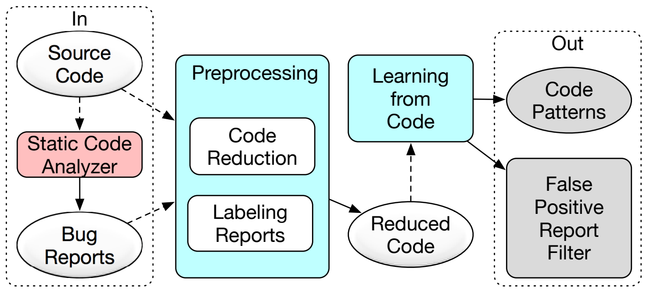


Figure 1. Learning approach overview.

the reduced version). Lastly, we use machine learning techniques on the reduced code to discover code structures that are correlated with false positive error reports and to learn a classifier that can filter out false positive error reports emitted by the SCA tool in the future.

2.1 Code Preprocessing

In prior work (Reynolds et al. 2017) we identified 14 core code patterns that lead to false positive error reports. We observed that in all code patterns the root cause of the false error report often spans over a small number of program locations. To better document these patterns, we performed manual code reduction to remove the parts of the code that were not related to the error report. After this manual reduction, the resulting program is effectively the smallest code snippet that still leads to the same false positive error report from the subject SCA tool.

In this work, we automated the code reduction step. Such reduction is important because code segments that are not relevant to the error report may introduce noise, causing over-fitting or spurious correlations. Now, we explain the reduction techniques we apply; method body and program slicing.

Method body. As a naive approach, we simply take the body of the method that contains the warning line in it (referred to as “warning method" later in the text). Note that, many of the code locations relevant for the error report are not inside the body of the warning method. In many cases, the causes of the report span over multiple methods and classes. Hence, this reduction is not a perfect way of isolating relevant code locations. However, if we can detect patterns in such sparse data, our models must be on the right track.

Program slicing. Given a fixed point in a program, program slicing is a technique that reduces that program to its minimal form, called slice, which still has the same behavior at that fixed point (Weiser 1981). The reduction is done by removing the code that does not affect the behavior at the given point. Computing the program slice from the warning line up to the entry point of a program would give us the backward slice which covers all code locations that are relevant for the error report (in theory). We will explain how we configured an industrial scale framework, WALA, to compute the backward slice later in Section 3 with more detail.

2.2 Learning

Filtering false positive error reports can be viewed as a binary classification problem with the classes True Positive and False Positive. In this binary classification problem, we have two major goals; 1) discovering code pieces correlated with these classes, and 2) learning a classifier (see Figure 1). Towards achieving these goals, we explore two different learning approaches. First, we use a simple Naive Bayes inference based learning model. Second, we use a neural network based language model called LSTM (Hochreiter and Schmidhuber 1997). The first approach is simple and interpretable. The second approach can learn more complex structures.

2.2.1 Naive Bayesian Inference

We formulate the problem as calculating the probability that an error report is either a true positive or a false positive, given the underlying code. So, the probability of the error report being a false positive is $P(e{=}0|code)$ where $e{=}0$ means there is no error in the code. Since there are only two classes, the probability of being a true positive can be simply computed as $P(e{=}1|code) = 1 - P(e{=}0|code)$. To calculate the probability $P(e{=}0|code)$, we use a simple Bayesian inference:

$$P(e=0|code) = \frac{P(code|e=0)P(e=0)}{P(code)} = \frac{P(code|e=0)P(e=0)}{P(code|e=0)P(e=0)+P(code|e=1)P(e=1)}$$

Where $P(e{=}0)$ and $P(e{=}1)$ are respectively the percentages of false positive and true positive populations in the dataset, and $P(code)$ is the probability of getting this specific code from the unknown distribution of all codes. To calculate $P(code|e{=}0)$ and $P(code|e{=}1)$, we formulate the code as a sequence of instructions (bytecodes), i.e., $code{=}< I_1, I_2, I_3, ..., I_n >$. So we rewrite $P(code|e{=}0)$ as,

$$\begin{aligned} P(code|e=0) &= P(I_1, I_2, ..., I_n|e=0) \\ &= P(I_1|e=0)P(I_2, ..., I_n|I_1, e=0) \\ &= P(I_1|e=0)P(I_2|I_1, e=0)P(I_3, ...I_n|I_1, I_2, e=0) \\ &\dots \\ &= P(I_1|e=0)P(I_2|I_1, e=0)...P(I_n|I_1, I_2, ..., e=0) \end{aligned}$$

To calculate each probability, we need to count the number of times each combination occurs in the dataset. However, for a complicated probability like $P(I_n|I_1, I_2, ..., e{=}0)$, we need to have a huge dataset to be able to estimate it accurately. In order to avoid this issue, we simplify this probability by assuming a Markov property. For this analysis, the Markov property simply means that the probability of seeing each instruction is independent of any other instruction in the code. Although this is not necessarily true in all codes, this assumption helps us build an initial model to have an intuition of what is happening in the dataset (we know this assumption is not likely for flow sensitive properties. Our second model does not need this assumption). With the Markov property, the underlying probability becomes:

$$P(code|e=0) = P(I_1|e=0)P(I_2|e=0)...P(I_n|e=0)$$

Calculating each of the $P(I_i|e{=}0)$ is very straightforward. We count the number of times instruction I_i appears in any false positive example, and we divide it by the total number of instructions in all of the false positive examples. Algorithm 1 shows how to calculate these probabilities. Line 3 counts the number of times each instruction appears in true positive and false positive examples.

Algorithm 1 Computing Probabilities

```
1: for each code C in Dataset do
2:     for each instruction I in C do
3:         $count[C.isTruePositive][I]{+}{+}$
4:         $total[C.isTruePositive]{+}{+}$
5:     end for
6: end for
7: for each instruction I do
8:     $P(I|e=1) \leftarrow count[True][I]/total[True]$
9:     $P(I|e=0) \leftarrow count[False][I]/total[False]$
10: end for
```

Then, line 4 counts the total number of instructions in each class. Finally, lines 8 and 9 compute the probabilities for instructions.

2.2.2 Long Short Term Memory

Long Short-Term Memory (LSTM) is a special kind of recurrent neural network introduced by Hochreiter and Schmidhuber (1997). The main motivation behind this model is to capture long-term dependencies in sequential data. Although there are many variations, a standard LSTM model typically has two recurrent connections; the output and the memory. The output recurrence is the norm of the recurrent neural networks. The memory, however, is specific to LSTM and it enables the model to remember information from the past (earlier tokens in the sequence), and by doing so, LSTM can capture long-term dependencies.

We think LSTM model is a good fit for our classification problem because 1) our data is sequential, and 2) it certainly has long-term dependencies which may be relevant to whether or not an error report is true or false. For instance, variable def-use pairs, method calls with arguments, accessing class fields are some of the program structures which would form long-term dependencies in code (see Section 3 for an example long-term dependency).

Carrier and Cho (2016) designed a single layer LSTM model for sentiment analysis which is also a binary classification problem like ours. In this work, we adopt this simple LSTM model using adadelta optimization algorithm (Zeiler 2012). To be able to make some observations by visualizing the inner workings of the model, we prefer having fewer (four) cells each of which is an LSTM (see Figure 4). Finding the optimum number of cells is not in the scope of this work. Following the LSTM layer, there is a pooling layer to compute a general representation of the data and finally logistic regression to classify the data into one of the two classes. Figure 2 shows the structure of the LSTM unrolled over time, with n being the length of the longest sequence.

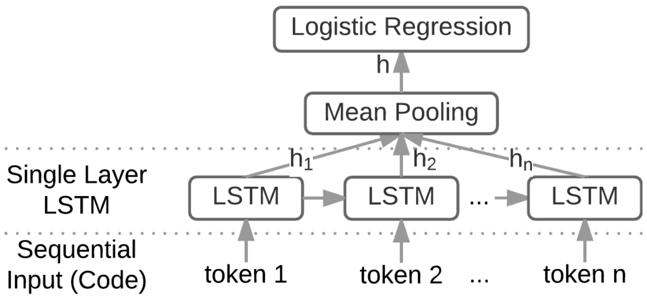


Figure 2. The LSTM model unrolled over time.

3. Case Study

This section presents the case study we conducted to evaluate the effectiveness of the proposed approach.

3.1 Subject SCA Tool and Warning Type

In this case study, we focus on the SQL injection flaw type. As the subject SCA tool, we use the FindSecBugs plug-in of FindBugs (Arteau 2016; Ayewah et al. 2008) (a widely-used tool for Java). This plug-in performs taint analysis to find SQL injection flaws. Very simply, taint analysis checks for data-flow from untrusted sources to safety critical sink points. For example, one safety-critical sink point for SQL injection is the "Connection.execute(String)" Java statement. A string parameter passed to this method is considered as tainted if it comes from an untrusted source such as an HTTP cookie or a user input (both are untrusted source because malicious users can leverage them to attack a system). FindSecBugs emits an SQL injection error report in such cases to warn the user of a potential security problem.

However, for complex source code, it may be difficult to determine whether the parameters are tainted. To give an example, it is a very common scenario to receive a user input to become a part of an SQL string. In such cases, developers often perform their own security checks against an injection threat. When the chain of information flow becomes too complicated, the SCA tool may not be able to correctly track it, and might, therefore, emit a false positive error report.

Note that, the proposed approach is not restricted to SQL injection flaws, FindSecBugs, or taint analysis which are just the subjects for this case study. Our learners do not make use of any piece of information that is specific to these subjects. Furthermore, we think that it can be possible to train a model that works for multiple flaw types or SCA tools.

3.2 Data

One of the biggest challenges for our problem is to find a sufficient dataset on which to train. We know of no publicly available benchmark datasets containing real-world programs with labeled error reports emitted by SCA tools. However, there are two benchmark test suites developed to evaluate the performance of SCA tools; Juliet (Boland and Black 2012) and Owasp benchmark (Owasp 2017). These test suites consist of test cases that exercise common weaknesses (Martin 2007). Note that, not all test cases really have an actual weakness. In fact, roughly half of the test cases are designed in ways that may trick SCA tools into emitting false reports. For this case study, we focused on the Owasp benchmark test suite as it has a bigger dataset for SQL injection flaw type. This dataset has 2,371 data points; 1,193 false positive and 1,178 true positive error reports.

Figure 3 shows an example Owasp test case for which FindSecBugs generates an error report. In line 7, the "param" variable gets a value from an HTTP header element which is considered to be a tainted source. The "param" variable is then passed to the "doSomething" method as an argument. In the "doSomething" method, starting at line 17, the tainted "param" argument is put into a HashMap object, line 21. Next, it is read back from the map into the "bar" variable in line 23. At this point, the "bar" variable has a tainted value. However, without doing anything with that tainted value, the program gets a new value from the map which is this time a hard-coded string, i.e., a trusted source. Finally "doSomething" returns this hard-coded string which gets concatenated into the "sql" variable at line 10. Then a callable statement is created and executed, lines 13 and 14. To summarize, the string concatenated with the SQL is hard coded and thus does not represent an injection threat. Therefore, the error report is a false positive.

3.3 Preprocessing

For practicality reasons, we focus on the bytecode representation of the data. With bytecode, there are fewer program specific tokens and syntactic components than that found in source code. Also, it is much easier for a machine learning model to work on the bytecode since it has already been simplified and itemized. In contrast, in the source code there might be multiple instructions in a single line and what each instruction is doing is, therefore, less easy to understand. This makes source code harder to analyze.

For the SQL injection dataset, we applied the two code reduction techniques (described in Section 2) leading to two different reduced datasets called "method body" and "backward slice" respectively. Application of method body reduction is straightforward, we simply take the bytecode for the body of the warning method. Backward slicing is more complex, as described below.

Tuning WALA. We use the WALA (IBM 2006) program slicer to

```java
public class BenchmarkTest16536 extends HttpServlet {
  public void doPost(HttpServletRequest r){
    String param = "";
    Enumeration<String> headers =
         r.getHeaders("foo");
    if (headers.hasMoreElements()) {
      // just grab first element
      param = headers.nextElement();
    }
    String bar = doSomething(param);
    String sql = "{call verifyUserPassword('foo','"
                 + bar + "')}";
    Connection con = DatabaseHelper.getConnection();
    CallableStatement stmt = con.prepareCall(sql);
    stmt.execute();
  } // end doPost

  private static String doSomething(String param){
    String bar = "safe!";
    HashMap<String,Object> map = new HashMap();
    map.put("keyA", "a_Value");
    map.put("keyB", param.toString());
    map.put("keyC", "another_Value");
    bar = (String) map.get("keyB");
    bar = (String) map.get("keyA");
    return bar;
  }
}
```

Figure 3. An example Owasp test-case that FindSecBugs generates a false positive error report for (simplified for presentation).

compute a backward slice with respect to a warning line. In theory, backward slice should cover all code locations related to the error report. However, program slicing is unsolvable in general and not scalable most of the time (Weiser 1981). In fact, we experienced excessive execution times when computing backward slices even for the simple short Owasp test cases. To avoid this problem, we configured WALA program slicer to narrow the scope and limit the amount of analysis it does for computing the slice.

First, we restricted the set of data dependencies by ignoring exception objects, base pointers, and heap components. We assume that exception objects are not relevant to the error report. Base pointers and heap variables, on the other hand, are just represented as indexes in the bytecode, over which our models cannot effectively reason, so we discarded them.

Second, we set the entry points as close to the warning method as possible. An entry point is the place where the backward slice ends. By default, this point would be the main method of the program. For the Owasp test suite, however, there is a large amount of code in the main method that is common for all test cases. Since this common code is unlikely to be relevant to any error reports, we rule it out by setting the warning method as the entry point for Owasp.

Third, we exclude Owasp utility classes, Java classes, and all classes of third party libraries as none of them are relevant to the error report for this case study. With this exclusion, we are not removing the references to these classes, just treating them as a black box. With the WALA tuning mentioned here, we are now able to compute a modified backward slice for Owasp test cases in reasonable times.

Note that, although WALA analyzes bytecode, the slice it outputs differs from bytecode with a few points. For presentation purposes, WALA uses some additional instructions like "new", "branch", "return", which do not belong to Java bytecode. Therefore, the dictionary of the method body dataset and the dictionary of the backward slice dataset are not exactly the same.

Now, we explain the further changes we performed for both datasets. First of all, we removed program-specific tokens and literal expressions because they may give away whether the error report is a true positive or a false positive. For the LSTM Classifier, we do this by deleting literal expressions and replacing program-specific objects with "UNK_OBJ" and method calls with "UNK_CALL". For the Naive Bayes Classifier, we do so by simply deleting them all. Note that, this step is also necessary to be able to generalize the classifier across programs. If we let the model learn from program-specific components, then it will not be able to do a good job on the code that does not share same components.

Lastly, for the Naive Bayes Classifier, we remove all arguments to instructions except the invoke instructions (invokeinterface, invokevirtual etc.). With them, we also keep the class names being called. This is done to simplify the dataset more. Furthermore, we treat all kinds of invoke instructions as the same by simply replacing them with "invoke". For the LSTM Classifier, we tokenized the data by whitespace. (e.g., with the invoke instructions, the instruction itself is one token and the class being invoked is one token). Therefore, when analyzing results, we use the word 'token' for LSTM and 'instruction' for Naive Bayes.

4. Results and Analysis

For all experiments, the dataset has been randomly split into 80% training set, and 20% test set. Table 1 summarizes the results. Accuracy is the percentage of correctly classified samples. Recall and precision are the percentages of correctly classified false positive samples with respect to all false positive samples and the samples classified as false positive, respectively. All three of the metrics are computed using the test portion of the datasets.

4.1 Naive Bayes Classifier Analysis

For the analysis of results, we consider any instruction I to be independent of SQL injection flaw if the value

$$\left[\frac{P(I|e=0)}{P(I|e=0)+P(I|e=1)} - 0.5\right]$$

is smaller than 0.1 in magnitude. We call this value "False Positive Dependence" and it ranges from -0.5 to 0.5 inclusive, where large positive values mean the instruction is correlated with false positive class and large negative values mean it is correlated with true positive class. Values around zero mean the instruction is equally likely to appear in both true positive and false positive classes (i.e. $P(I|e{=}0) \simeq P(I|e{=}1)$) and therefore is independent of SQL injection flaw.

We started the experiment by running the Naive Bayes Classifier on the method body dataset. Although the accuracy result is not very high for this experiment (63%, in Table 1), it confirmed that the bytecode contains recognizable signals indicating false positive error reports.

The Naive Bayes Classifier learns the conditional property of each instruction, given that an error exists. We observed that instructions like iload, ifne, etc., are equally likely to appear in both true and false positive samples. Therefore, their "False Positive Dependence" value is below the threshold (0.1) and these instructions are independent of SQL injection flaws.

classifier	dataset	training time (m)	recall	precision %	accuracy
Naive Bayes	method body	0.02	60	64	63
	backward slice	0.03	66	75	72
LSTM	method body	17	81.3	97.3	89.6
	backward slice	18	97	78.2	85

Table 1. Results.

instructions	False Positive Dependence	
	method body	backward slice
invoke esapi.Encoder	-0.09	-0.36
invoke java.util.ArrayList	0.04	0.18
invoke java.util.HashMap	0.18	0.25

Table 2. Important instructions

Next, the only instruction we found to be correlated with the false positive class is “invoke java.util.HashMap" (Table 2). By manually examining the Owasp test suit we see that, it is a common pattern to insert a tainted string and also a safe string into a HashMap, and then extract the safe string from the HashMap to become a part of an SQL (see Figure 3 for an example). This is done to “trick” SCA tools into emitting incorrect reports, and Naive Bayes Classifier correctly identifies this situation.

For the second experiment, we did the same analysis for the backward slices dataset. Our hypothesis is that analyzing only the code available in method bodies is not enough. Instead, we need to consider all relevant instructions, even if they are outside the method body. Backward slices provide this information by including instructions that may be relevant to the error report.

Running the Naive Bayes model on the backward slice dataset confirms our findings in method body dataset. In addition to HashMap, this model also learns from the backward slices that ArrayList invocation is highly correlated with false positives, and Encoder invocation is highly correlated with true positives. The False Positive Dependence for significant classes invoked is shown in Table 2. These correlations can easily be justified by examining the code.

In Owasp, ArrayList is used to trick the analyzer, much like HashMap did in the previous discussion. Furthermore, Encoder is mainly used in the dataset for things like HTML encoding but not for SQL escaping. This pattern is used to trick the analyzer into missing some true positive samples, which our model identifies as well.

The main reason for improved results in backward slices dataset is that we have access to all relevant instructions and irrelevant instructions which act as noise have been removed. This increases the confidence of the classifier. Table 2 shows that dependence values have increased in magnitude for backward slices, which means the classifier is more confident in their effect on the code.

Weaknesses. To better understand the limitations of the Bayes model, we examined some of the incorrectly classified examples. We observed that the model can still identify the role of each instruction correctly. However, in those examples, multiple instructions are correlated with true positives and a single or a few instruction that make the string safe. By it’s nature, the Naive Bayes model cannot take into account that a single instruction is enough to make the string untainted. So when we multiply the probabilities for all instruction, that single instruction cannot make much of a difference and the Bayes model end up classifying the code incorrectly.

4.2 LSTM Classifier Analysis

With the LSTM classifier, we achieved 89.6% and 85% accuracy for the method body and backward slice datasets respectively (Table 1). The classifier trained on method body dataset is very precise, i.e. 97.3% of the error reports classified as false positive are indeed false positive. However, this classifier misses 18.7% of false positives, i.e., classifying them as true positives. The situation is reversed for the classifier trained on the backward slice dataset. It catches 97% of the false positives but also filters out many true positive reports, i.e., 21.8% of the samples classified as false positive are indeed true positive.

We examined a sample program in which the classifier trained on method body dataset can classify correctly but the classifier trained on backward slice dataset can not. We observed that many instructions that only exist in the method body dataset, like “aload_0”, “i_const_0”, “dup” etc., are found to be important by the classifier. Note that, these instructions are not in the backward slice dataset either because they do not have any effect on the warning line, or because of the tuning we did. The first case, learning the instructions that are not related to the warning line, is basically over-fitting the noise. The second case, however, requires a deeper examination that we defer to the future work. Just relying on the first case, we think that the classifier trained on the backward slice dataset is more generalizable as this dataset has lesser noise and more report relevant components. Hence, in the rest of this section, we will only analyze the classifier trained on the backward slice dataset.

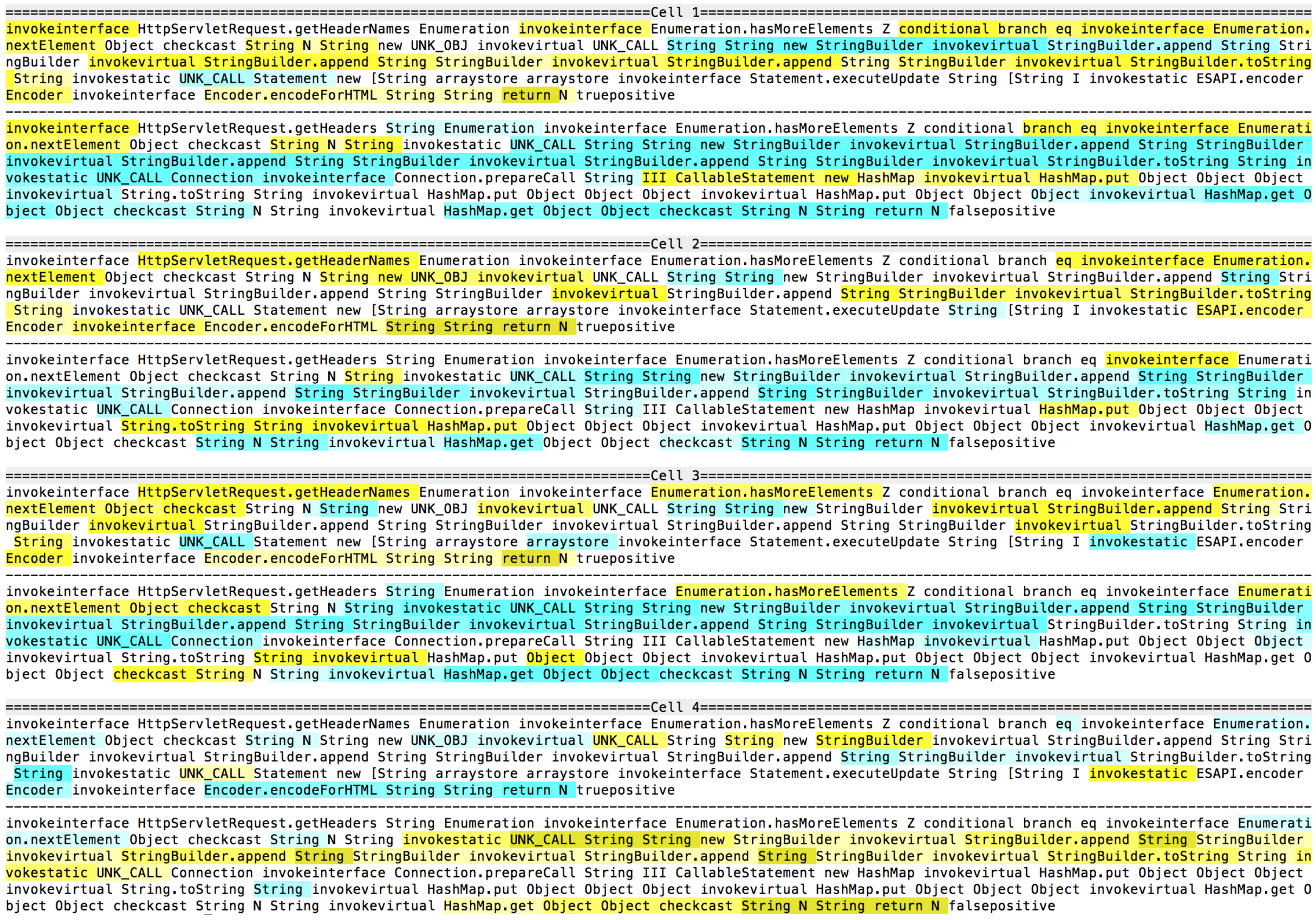
Understanding the source of the LSTM’s high performance is very challenging as we cannot fully unfold the inner workings of it. Nevertheless, we can visualize the output values of some cells as suggested by Karpathy et al. (2015). Figure 4 illustrates output values of four cells for two correctly classified backward slices by coloring the background. The latter is the slice computed for the false positive sample in Figure 3. The former is the slice computed for a true positive which is structurally similar to the latter with two important differences; 1) “doSomething" method is defined in an inner class, and 2) an HTML encoder is called in “doSomething" method (instead of HashMap operations). Note that, last tokens are the labels; “truepositive" and “falsepositive".

Now, we discuss some interesting observations from Figure 4. First, due to the memory component of LSTM, the background color of a token (output for that token) does not solely depend on that token but is affected by the history. For example, looking at the Cell 1, the first token causes some yellowness in both samples. Since there is no history before the first token, this yellowness is solely due to that token. On the other hand, looking at Cell 4, the first lines of both samples mostly the same except that there is one token with cyan background in the true positive sample; “eq". The only difference in the first lines is the tainted sources invoked which are “HttpServletRequest.getHeaderNames" and “HttpServletRequest.getHeaders String" in true positive and false positive respectively. Therefore, the only reason why “eq" token is interesting only in the true positive sample must be the invocation of the tainted source “HttpServletRequest.getHeaderNames". This is a good example of a long-term dependency as there are six other tokens in between.

Second, in both samples, all cells have a high output for the “Enumeration.nextElement" token, which is highly relevant for the error report as it is the tainted source. Note that, all cells treat this token the same way in both samples. On the other hand, for the last return instructions in both samples all cells have very high output, but this time the output is negative in the true positive sample and positive in the false positive sample (in first three cells and vice versa for in the last cell) which clearly happens so due to the history of tokens. This is a good example illustrating the LSTM’s ability to infer the context.

Next, looking at the false positive sample in Figure 3, we see that the core reason of falseness resides in the body of the “doSomething" method. In particular, HashMap put and get instructions are very critical. We see that all cells have very high output for a subset of the tokens that correspond to that instructions. These high output values for HashMap put and get instructions match the findings of the Bayesian model. Furthermore, all cells go very yellow for the “Encoder.encodeForHTML" method call in the true positive sample. These high results for “Encoder" tokens are also in accordance with the findings of the Bayesian model.

Lastly, Figure 4 shows that although most of the high output values are reasonable and interpretable, there are still many that we



```
=============================================================================Cell 1============================================================================
invokeinterface HttpServletRequest.getHeaderNames Enumeration invokeinterface Enumeration.hasMoreElements Z conditional branch eq invokeinterface Enumeration.
nextElement Object checkcast String N String new UNK_OBJ invokevirtual UNK_CALL String String new StringBuilder invokevirtual StringBuilder.append String Stri
ngBuilder invokevirtual StringBuilder.append String StringBuilder invokevirtual StringBuilder.append String StringBuilder invokevirtual StringBuilder.toString
 String invokestatic UNK_CALL Statement new [String arraystore arraystore invokeinterface Statement.executeUpdate String [String I invokestatic ESAPI.encoder
Encoder invokeinterface Encoder.encodeForHTML String String return N truepositive
--------------------------------------------------------------------------------------------------------------------------------------------------------------
invokeinterface HttpServletRequest.getHeaders String Enumeration invokeinterface Enumeration.hasMoreElements Z conditional branch eq invokeinterface Enumerati
on.nextElement Object checkcast String N String invokestatic UNK_CALL String String new StringBuilder invokevirtual StringBuilder.append String StringBuilder
invokevirtual StringBuilder.append String StringBuilder invokevirtual StringBuilder.append String StringBuilder invokevirtual StringBuilder.toString String in
vokestatic UNK_CALL Connection invokeinterface Connection.prepareCall String III CallableStatement new HashMap invokevirtual HashMap.put Object Object Object
invokevirtual String.toString String invokevirtual HashMap.put Object Object Object invokevirtual HashMap.put Object Object Object invokevirtual HashMap.get O
bject Object checkcast String N String invokevirtual HashMap.get Object Object checkcast String N String return N falsepositive

=============================================================================Cell 2============================================================================
invokeinterface HttpServletRequest.getHeaderNames Enumeration invokeinterface Enumeration.hasMoreElements Z conditional branch eq invokeinterface Enumeration.
nextElement Object checkcast String N String new UNK_OBJ invokevirtual UNK_CALL String String new StringBuilder invokevirtual StringBuilder.append String Stri
ngBuilder invokevirtual StringBuilder.append String StringBuilder invokevirtual StringBuilder.append String StringBuilder invokevirtual StringBuilder.toString
 String invokestatic UNK_CALL Statement new [String arraystore arraystore invokeinterface Statement.executeUpdate String [String I invokestatic ESAPI.encoder
Encoder invokeinterface Encoder.encodeForHTML String String return N truepositive
--------------------------------------------------------------------------------------------------------------------------------------------------------------
invokeinterface HttpServletRequest.getHeaders String Enumeration invokeinterface Enumeration.hasMoreElements Z conditional branch eq invokeinterface Enumerati
on.nextElement Object checkcast String N String invokestatic UNK_CALL String String new StringBuilder invokevirtual StringBuilder.append String StringBuilder
invokevirtual StringBuilder.append String StringBuilder invokevirtual StringBuilder.append String StringBuilder invokevirtual StringBuilder.toString String in
vokestatic UNK_CALL Connection invokeinterface Connection.prepareCall String III CallableStatement new HashMap invokevirtual HashMap.put Object Object Object
invokevirtual String.toString String invokevirtual HashMap.put Object Object Object invokevirtual HashMap.put Object Object Object invokevirtual HashMap.get O
bject Object checkcast String N String invokevirtual HashMap.get Object Object checkcast String N String return N falsepositive

=============================================================================Cell 3============================================================================
invokeinterface HttpServletRequest.getHeaderNames Enumeration invokeinterface Enumeration.hasMoreElements Z conditional branch eq invokeinterface Enumeration.
nextElement Object checkcast String N String new UNK_OBJ invokevirtual UNK_CALL String String new StringBuilder invokevirtual StringBuilder.append String Stri
ngBuilder invokevirtual StringBuilder.append String StringBuilder invokevirtual StringBuilder.append String StringBuilder invokevirtual StringBuilder.toString
 String invokestatic UNK_CALL Statement new [String arraystore arraystore invokeinterface Statement.executeUpdate String [String I invokestatic ESAPI.encoder
Encoder invokeinterface Encoder.encodeForHTML String String return N truepositive
--------------------------------------------------------------------------------------------------------------------------------------------------------------
invokeinterface HttpServletRequest.getHeaders String Enumeration invokeinterface Enumeration.hasMoreElements Z conditional branch eq invokeinterface Enumerati
on.nextElement Object checkcast String N String invokestatic UNK_CALL String String new StringBuilder invokevirtual StringBuilder.append String StringBuilder
invokevirtual StringBuilder.append String StringBuilder invokevirtual StringBuilder.append String StringBuilder invokevirtual StringBuilder.toString String in
vokestatic UNK_CALL Connection invokeinterface Connection.prepareCall String III CallableStatement new HashMap invokevirtual HashMap.put Object Object Object
invokevirtual String.toString String invokevirtual HashMap.put Object Object Object invokevirtual HashMap.put Object Object Object invokevirtual HashMap.get O
bject Object checkcast String N String invokevirtual HashMap.get Object Object checkcast String N String return N falsepositive

=============================================================================Cell 4============================================================================
invokeinterface HttpServletRequest.getHeaderNames Enumeration invokeinterface Enumeration.hasMoreElements Z conditional branch eq invokeinterface Enumeration.
nextElement Object checkcast String N String new UNK_OBJ invokevirtual UNK_CALL String String new StringBuilder invokevirtual StringBuilder.append String Stri
ngBuilder invokevirtual StringBuilder.append String StringBuilder invokevirtual StringBuilder.append String StringBuilder invokevirtual StringBuilder.toString
 String invokestatic UNK_CALL Statement new [String arraystore arraystore invokeinterface Statement.executeUpdate String [String I invokestatic ESAPI.encoder
Encoder invokeinterface Encoder.encodeForHTML String String return N truepositive
--------------------------------------------------------------------------------------------------------------------------------------------------------------
invokeinterface HttpServletRequest.getHeaders String Enumeration invokeinterface Enumeration.hasMoreElements Z conditional branch eq invokeinterface Enumerati
on.nextElement Object checkcast String N String invokestatic UNK_CALL String String new StringBuilder invokevirtual StringBuilder.append String StringBuilder
invokevirtual StringBuilder.append String StringBuilder invokevirtual StringBuilder.append String StringBuilder invokevirtual StringBuilder.toString String in
vokestatic UNK_CALL Connection invokeinterface Connection.prepareCall String III CallableStatement new HashMap invokevirtual HashMap.put Object Object Object
invokevirtual String.toString String invokevirtual HashMap.put Object Object Object invokevirtual HashMap.put Object Object Object invokevirtual HashMap.get O
bject Object checkcast String N String invokevirtual HashMap.get Object Object checkcast String N String return N falsepositive
```

Figure 4. LSTM color map for two correctly classified slices. First slice is a true positive and second slice is the false positive in Figure 3. Cyan, yellow, and white background mean positive, negative and under threshold (± 0.35) output value respectively. There is only one tint of white, but for cyan and yellow, the darker the background the larger the output is (in magnitude).

cannot explain. This situation is common with neural networks and we will continue to explore it in the future work.

4.3 Threats to Validity

Like any empirical study, our findings are subject to threats to internal and external validity. For this case study, we were primarily concerned with threats to external validity since they limit our ability to generalize the results of our studies.

For this case study, all majors generalizability threats are related to the representativeness of the datasets. First, Owasp test cases are not truly representative of real-world programs. They are not large in size and they do not handle any particular functionality other than exercising the targeted weakness. Nevertheless, they are still a good starting point for our problem. Second, the datasets only cover one type of flaw, SQL injection, emitted by one SCA tool, FindSecBugs. However, FindSecBugs is a plug-in of FindBugs which we think is a good representative of open source SCA tools. Next, we see that FindSecBugs performs the same analysis all other types of security flaws it checks (such as command injection, LDAP injection, and XSS). Therefore, FindSecBugs and SQL injection flaw is a good combination representing security flaws and checkers. Lastly, we only experiment with Java bytecode. We will work to address these threats in future work (see Section 6).

5. Related Work

There are three threads of related work.

Code reduction. We looked into three code reduction techniques; delta debugging (Zeller and Hildebrandt 2002), hierarchical delta debugging (Misherghi and Su 2006), and c-reduce Regehr et al. (2012). Delta debugging isolates the failure-inducing input by applying a systematic binary search based reduction to the input that causes the program to crash. Hierarchical delta debugging has the same goal but it specifically targets structured inputs. C-reduce is a tool that applies a set of C/C++ specific transformations for finding small programs that cause compiler crashes.

All of these techniques use a user-provided interestingness criteria to guide the reduction. For our problem, writing such an interestingness criteria is not possible because in general, falseness of a report cannot be automatically detected. Thus, we choose not to use these reduction techniques.

Spurious error report filtering. Previous research has looked into false positive error report problem of SCA tools. Kremenek and Engler (2003) propose the z-ranking technique to rank error reports emitted by SCA tools using statistical analysis based on the likelihood of being a real bug. Kim et al. (2010) report 70% reduction in the number of false buffer overflow error reports for C code by applying an iterative analysis with different abstraction levels. Yüksel and Sözer (2013) experiment with 34 different machine learning algorithms to classify SCA error reports using ten artifact characteristics and report 86% accuracy in classification. Tripp et al. (2014) propose a customizable approach for learning a classifier for false error reports produced with taint analysis for JavaScript.

None of these approaches apply machine learning to the source code. We differ from them in this respect.

Natural language processing (NLP) techniques applied to code. NLP techniques work very well in practice due to the fact that the language humans use in the daily life is repetitive and predictable. Hindle et al. (2012) argue source code written by human developers is similarly repetitive and predictable. Therefore, NLP techniques can potentially be very successful in learning properties from code. Since the work of Hindle et al. (2012), there have been many studies using statistical language models for handling important software development and maintenance tasks (Nguyen et al. 2013; Tu et al. 2014; Raychev et al. 2014, 2015; Allamanis et al. 2015, 2016; Fowkes and Sutton 2016; Gu et al. 2016; White et al. 2016; Dam et al. 2016).

Although these work represent a good example of successful application of NLP techniques for source code processing, none of them solves the false error report classification problem.

6. Conclusion and Future Work

We presented a learning approach to find program structures that cause the state of the art SCA tools to emit false error reports and filter out such false error reports with a classifier trained on the code. In particular, we used a Naive Bayes model and an LSTM model. With the Bayes model, we discovered interesting signals involving Java collection objects. Investigating these correlations, we found that FindSecBugs, the subject SCA tool, cannot successfully reason about very simple usage scenarios of Java collection objects like “HashMap" and “List". Therefore some false positive results might be avoided if SCA developers improve their analysis of collection classes in future. With the LSTM model, we achieved 89.6% and 85% accuracy in classification for the method body and the backward slice datasets respectively. By coloring the background of the input tokens based on the output of LSTM cells, we 1) showed a long-term dependency in the data, 2) demonstrated LSTMs' capability of inferring the context, and 3) showed how LSTMs output values agree with the findings of Naive Bayes model.

In future work, we will aim at expanding our datasets to improve the generalizability of our approach. To build a dataset of real-world programs with true and false positive error reports generated for them, we will use crowd-sourcing techniques as studied by LaToza and Hoek (2016) using checklist questions to review SCA results (Ayewah and Pugh 2009).

After improving our datasets, we will conduct extended empirical studies including more SCA tools and flaw types and using more advanced machine learning techniques. In particular, we are interested in investigating recursive neural networks (Socher et al. 2011) for learning on abstract syntax tree (AST) representation of the code. One downside of the bytecode representation we used in this work is that there is not contextual information in it. In contrast, AST representation can provide this information which advanced learning techniques can potentially leverage.

Lastly, we will extend the approach to become a semi-supervised incremental online service that SCA developers and users can leverage to improve the SCA tools quality and practicality.

Acknowledgments

This work was sponsored by the Department of Homeland Security under grant #D15PC00169. We like to thank our partners at IUPUI, James H. Hill, Abhinandan B. Jayanth, Rajeev R. Raje, and Zachary Paul Reynolds, for their insights and feedback throughout this project.

References

M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting Accurate Method and Class Names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786849.

M. Allamanis, H. Peng, and C. Sutton. A Convolutional Attention Network for Extreme Summarization of Source Code. *arXiv:1602.03001 [cs]*, Feb. 2016.

P. Arteau. Find security bugs, 2016. URL `http://find-sec-bugs.github.io`.

N. Ayewah and W. Pugh. Using Checklists to Review Static Analysis Warnings. In *Proceedings of the 2Nd International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, DEFECTS '09, pages 11–15, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-654-0. doi: 10.1145/1555860.1555864.

N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE software*, 25(5), 2008.

T. Boland and P. E. Black. Juliet 1.1 c/c++ and java test suite. *Computer*, 45 (10):0088–90, 2012.

P.-L. Carrier and K. Cho. Find security bugs, 2016. URL `http://deeplearning.net/tutorial/lstm.html`.

H. K. Dam, T. Tran, J. Grundy, and A. Ghose. DeepSoft: A vision for a deep model of software. *arXiv:1608.00092*, July 2016.

J. Fowkes and C. Sutton. Parameter-free Probabilistic API Mining Across GitHub. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 254–265, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950319.

X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep API Learning. *arXiv:1605.08535 [cs]*, May 2016.

A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering*, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735, Nov. 1997. ISSN 08997667.

IBM. The t.j.watson libraries for analysis (wala), 2006. URL `http://wala.sourceforge.net/wiki/index.php`.

A. Karpathy, J. Johnson, and L. Fei-Fei. Visualizing and Understanding Recurrent Networks. *arXiv:1506.02078*, June 2015.

Y. Kim, J. Lee, H. Han, and K.-M. Choe. Filtering false alarms of buffer overflow analysis using SMT solvers. *Information and Software Technology*, 52(2):210–219, Feb. 2010. ISSN 0950-5849. doi: 10.1016/j.infsof.2009.10.004.

T. Kremenek and D. Engler. Z-ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 295–315, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 978-3-540-40325-8.

T. D. LaToza and A. v. d. Hoek. Crowdsourcing in Software Engineering: Models, Motivations, and Challenges. *IEEE Software*, 33(1):74–80, Jan. 2016. ISSN 0740-7459. doi: 10.1109/MS.2016.12.

R. A. Martin. Common weakness enumeration. *Mitre Corporation*, 2007. URL `https://cwe.mitre.org`.

G. Misherghi and Z. Su. Hdd: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*, pages 142–151. ACM, 2006.

T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A Statistical Semantic Language Model for Source Code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 532–542, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491458.

F. Owasp. Open web application security project, 2017. URL `https://www.owasp.org/index.php/Benchmark`.

V. Raychev, M. Vechev, and E. Yahav. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594321.

V. Raychev, M. Vechev, and A. Krause. Predicting Program Properties from "Big Code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2677009.

J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for c compiler bugs. In *ACM SIGPLAN Notices*, volume 47, pages 335–346. ACM, 2012.

Z. Reynolds, A. Jayanth, U. Koc, A. Porter, R. Raje, and J. Hill. Identifying and documenting false positive patterns generated by static code analysis tools. In *4th International Workshop On Software Engineering Research And Industrial Practice*, 2017.

R. Socher, C. C. Lin, C. Manning, and A. Y. Ng. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 129–136, 2011.

O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin. ALETHEIA: Improving the Usability of Static Security Analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 762–774, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2957-6. doi: 10.1145/2660267.2660339.

Z. Tu, Z. Su, and P. Devanbu. On the Localness of Software. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 269–280, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635875.

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

M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep Learning Code Fragments for Code Clone Detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 87–98, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3845-5. doi: 10.1145/2970276.2970326.

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

M. D. Zeiler. ADADELTA: An Adaptive Learning Rate Method. *arXiv:1212.5701 [cs]*, Dec. 2012.

A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.