Striking a Balance: Pruning False-Positives from Static Call Graphs

Akshay Utture
University of California, Los Angeles
U.S.A.
akshayutture@ucla.edu

Shuyang Liu
University of California, Los Angeles
U.S.A.
sliu44@cs.ucla.edu

Christian Gram Kalhauge
DTU
Denmark
chrg@dtu.dk

Jens Palsberg
University of California, Los Angeles
U.S.A.
palsberg@ucla.edu

ABSTRACT

Researchers have reported that static analysis tools rarely achieve a false-positive rate that would make them attractive to developers. We overcome this problem by a technique that leads to reporting fewer bugs but also much fewer false positives. Our technique prunes the static call graph that sits at the core of many static analyses. Specifically, static call-graph construction proceeds as usual, after which a call-graph pruner removes many false-positive edges but few true edges. The challenge is to strike a balance between being aggressive in removing false-positive edges but not so aggressive that no true edges remain. We achieve this goal by automatically producing a call-graph pruner through an automatic, ahead-of-time learning process. We added such a call-graph pruner to a software tool for null-pointer analysis and found that the false-positive rate decreased from 73% to 23%. This improvement makes the tool more useful to developers.

CCS CONCEPTS

• **Software and its engineering** → **Automated static analysis**; • **Computing methodologies** → Supervised learning by classification.

ACM Reference Format:
Akshay Utture, Shuyang Liu, Christian Gram Kalhauge, and Jens Palsberg. 2022. Striking a Balance: Pruning False-Positives from Static Call Graphs. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510166

1 INTRODUCTION

The Problem. Christakis and Bird [14] interviewed developers about program analysis tools and they concluded:

> *Program analysis design should aim for a false-positive rate no higher than 15–20%.*

Other empirical studies have found similar results [6, 25, 40]. Until now, this goal has been particularly hard to achieve for *static* analyses, which are tools that analyze programs without executing them.

As a motivating experiment, we tried Wala [47], which is one of the best tools for static analysis of Java bytecode, on a subset of the NJR-1 benchmark suite [35]. For each benchmark, we compared the edges in the static call graph with the edges found by executing the benchmark. With a context-insensitive analysis, Wala has a false-positive rate of 76%, while with a better but also much slower context-sensitive analysis, the false-positive rate is 70%. Those results are disappointing though we must emphasize that call graphs are usually fed to client tools rather than directly to developers. So, we did a second experiment to see how the high false-positive rate of call-graphs affects client tools. Specifically, we implemented a version of a static analysis for warning about null-pointer problems [21] that is a client of the context-insensitive call graphs produced by Wala. We ran this tool on the same subset of NJR-1 and again had disappointing results: 60 bugs among 223 warnings, on average, so a false-positive rate of 73%. We can easily imagine how a developer will tire of investigating warnings that in nearly three of every four cases are false alarms. The false alarms have several causes, but an important cause is the high false-positive rate in the underlying static call graph. Hence, we can also see a glimmer of hope: if we can reduce the false-positive rate of static call-graph constructors, we may be able to move client tools closer to the goal of a false-positive rate of 15–20%.

Our Idea. Our approach stems from another conclusion by Christakis and Bird [14] who reported a preference of developers:

> *When forced to choose between more bugs or fewer false positives, they typically choose the latter.*

This quote inspired our idea for how to improve the false-positive rate: we will report *fewer* bugs but also *much fewer* false positives. Indirect support for this idea comes from previous work that showed that practical static analyses aren't totally sound [31, 43] and therefore may miss bugs. Thus, developers expect bug reports to be incomplete so reporting fewer bugs seems acceptable.

We want to reduce the false-positive rate in a modular way that leaves existing call-graph constructors unchanged. This brings us to our idea of a *call-graph pruner* that statically post-processes a static call graph by removing *many* false-positive edges but *few* true edges. The challenge is to strike a balance between being aggressive in removing false-positive edges but no so aggressive that no true edges remain. Additionally, we have to do better than removing edges at random because random removals will leave the false-positive rate unchanged.

How can we design a call-graph pruner?

Our Approach. We execute an automatic, ahead-of-time learning process on results from both a static and a dynamic call-graph constructor. The outcome is a call-graph pruner that works as follows. The call-graph pruner determines the probability that an edge in the call graph is a false positive, and if this probability is above a threshold, then the call-graph pruner removes the edge. We can vary this threshold and thereby tune the call-graph pruner.

In contrast to previous work on using a dynamic analysis to improve a static analysis [3, 13, 16], we use the dynamic call-graph constructor only in an ahead-of-time training phase and only on a training set of programs. Once the training phase has produced a call-graph pruner, the combination of the call-graph constructor and the call-graph pruner is itself a static analysis, as illustrated in Figure 1.

Our Contributions and the Rest of the Paper. We begin with an example of how a call-graph pruner works (Section 2) and then we detail our contributions:

- We present the design (Section 3) and implementation (Section 4) of a tool that produces call-graph pruners.
- We show experimentally (Section 5) that adding a call-graph pruner to a client tool can significantly decrease the false-positive rate, in one case from 73% to 23%. Specifically, we added a call-graph pruner to the tool for warning about null-pointer problems, after which we got 15 bugs among 20 warnings, on average. Thus we reported 45 fewer bugs but also 158 fewer false positives.
- We show experimentally (Section 5) that the overhead of adding a call-graph pruner is 18% of the original call-graph analysis time.

We end with a discussion of related work (Section 6) and our conclusion (Section 7).

Significance. Call-graph pruners improve static call-graphs significantly and thereby make client tools more useful to developers.

2 EXAMPLE

Now we give an example of a call-graph pruner, how it works on a example call graph, and how it affects a client analysis for warning about null-pointer problems. Our example program in Figure 2, shown in full in the Appendix, has three classes `A`, `B`, `C`, each of which has a method `foo`, and a main method that contains a method call `x.foo(x.f)`. The call to `getObjC()` returns an object of type `C`, which is then assigned to the variable `x`. On the next line, the access `x.f` happens, but the field `A.f` may be uninitialized hence null. Thus the call `x.foo(x.f)` may pass null as an argument to `C.foo`, which, in turn, at the call `c.toString()`, may throw a NullPointerException. The program has two additional methods, including `getObjC`, that we omitted from Figure 2.

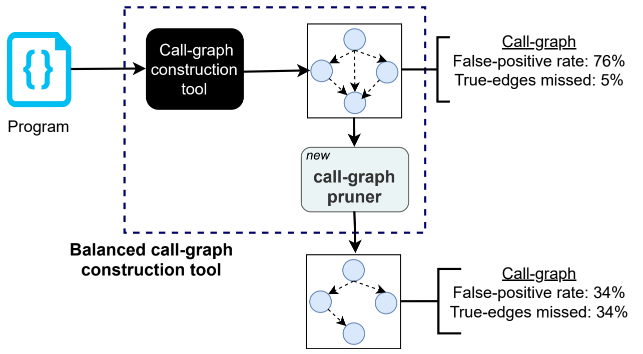
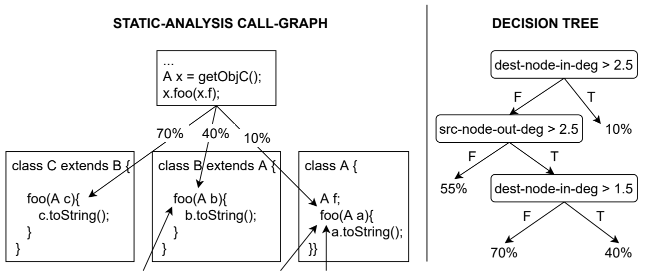


Figure 1: Overview of our technique



```
...
A x = getObjC();
x.foo(x.f);
```

```
class C extends B {
  foo(A c){
    c.toString();
  }
}
```

```
class B extends A {
  foo(A b){
    b.toString();
  }
}
```

```
class A {
  A f;
  foo(A a){
    a.toString();
}}
```

Figure 2: Example call graph and call-graph pruner

Null-Pointer Warnings. As we mentioned in Section 1, we implemented a version of a static analysis for warning about null-pointer problems. This analysis finds null-pointer problems that stem from uninitialized fields, like the problem with `c.toString()` that is caused by the uninitialized field `A.f`. If we run this tool on the example program, we get *three* warnings, one for each call of `toString` in the `foo` methods. One of them is a true warning but the other two are false alarms. Let us investigate how that could happen and what a call-graph pruner can do about it.

Call Graph. The null-pointer tool uses a static call-graph constructor that built the call graph shown in Figure 2. In a call graph, each node is a method, and each edge is a directed edge from one method to another. Such an edge represents a call that *may happen* during the execution of the program.

The call-graph constructor uses a data-flow analysis to analyze the entire program, including the methods that we omitted from Figure 2. We skip the details of how this works and instead we focus on the constructed call graph. Specifically, in Figure 2 we focus on the four nodes for the main method, `A.foo`, `B.foo`, and `C.foo`. The call graph has an edge from the main method to each of `A.foo`, `B.foo`, and `C.foo`, as well as an edge some other method to `B.foo` and a couple of edges from some other methods to `A.foo`. The edge from main to `C.foo` is a true edge, while the edges from main to `A.foo` and from main to `B.foo` are false positives.

The false call-graph edges from main to each of `A.foo` and `B.foo` can arise from difficult-to-analyze methods, one of which is part of the full example program in the appendix.

The Null-Pointer Analysis in more Detail. Based on the call graph in Figure 2, the null-pointer analysis derives that `x.foo(x.f)` may call any of `A.foo`, `B.foo`, and `C.foo`. Then the null-pointer analysis uses the rule that

> if a field is not initialized by the end of a constructor, it is marked as *Uninitialized*; and if an *Uninitialized* field is dereferenced, the analysis gives a null-pointer warning.

Thus, the analysis concludes that each of the `foo` methods may be passed null as an argument, and thus it issues a warning for every one of those methods.

Call-Graph Pruner. The goal of a call-graph pruner is to remove edges from the call-graph, preferably many false-positive edges and few true edges. The key component of a call-graph pruner is a classifier that computes the probability that a call-graph edge is a true-positive. Based on that probability, a call-graph pruner will decide whether to keep or to remove the edge. Figure 2 shows a classifier that is represented as a decision tree. Each internal node of the decision tree asks a true-false question about a call-graph edge. The recursive decision process begins in the root of the decision tree; if the answer to the question at the root is false, we move to the left subtree, while if the answer is true, we move to the right subtree. When we reach a leaf, we find the probability that the call-graph edge is a true-positive. The probabilities computed for each call-graph edge in this fashion are marked on the call graph in Figure 2. Based on these probabilities, we will decide whether to keep or remove the call-graph edge.

The decision tree in Figure 2 has three internal nodes that are labeled with questions about *dest-node-in-deg*, which is the in-degree of the destination node of the edge, and about *src-node-out-deg*, which is the out-degree of the source node of the edge. For example, the edge from main to `C.foo` has destination-node in-degree 1 and source-node out-degree 3. This gives us the path false-true-false, which assigns the edge the probability 70%. Similarly, the edges from main to `A.foo` and `B.foo` get probabilities 10% and 40%, respectively. The call graph in Figure 2 shows those three probabilities.

Let us set a threshold of 50% for when we deem an edge to be a false-positive: if the probability of being a true-positive is below 50%, we remove the edge. Then the call-graph pruner will remove the edges from main to `A.foo` and `B.foo`. Hence, the null-pointer analysis will issue just a single warning, and indeed a true warning, namely for the call of `toString` in `C.foo`.

3 CALL-GRAPH PRUNERS

Now we describe how we use machine learning to produce a call-graph pruner.

3.1 Overview

We will use Program to denote the set of Java bytecode programs.

A call graph $G \in$ CallGraph is a multi-graph in which each node represents a method and each edge represents a potential transfer of control at a method call. Two nodes can have multiple edges between them because of multiple method calls. Each edge has a label that identifies the method call site.

We distinguish between two kinds of call-graph constructors that have the same type:

$$\begin{aligned}\mathsf{StaticCallGraphConstructor} &= \mathsf{Program} \rightarrow \mathsf{CallGraph}\\ \mathsf{DynamicCallGraphConstructor} &= \mathsf{Program} \rightarrow \mathsf{CallGraph}\end{aligned}$$

Here, an element of StaticCallGraphConstructor constructs a call graph without running the program, while, in contrast, an element of DynamicCallGraphConstructor runs an instrumented version of the program on one or more inputs and examines the output from the instrumentation.

The key component of each call-graph pruner is a classifier. A classifier $C \in$ Classifier is a function that maps a vector of feature values for an edge to a probability that the edge is a true-positive. In our case, such a vector has 11 elements that we will define in Section 3.3.

Our tool for generating classifiers implements a function of this type:

$$\begin{aligned}\text{classifier generator} \quad : \quad & (\mathsf{StaticCallGraphConstructor} \times\\ & \mathsf{DynamicCallGraphConstructor} \times\\ & \mathsf{Set[Program]})\\ & \rightarrow \mathsf{Classifier}\end{aligned}$$

Our classifier generator executes an automatic, ahead-of-time learning process on results from running both a static and a dynamic call-graph constructor on a training set of programs. The dynamic call graphs serve as ground-truth for the learning process. We will detail this learning process in Section 3.2.

Once we have a classifier, we can use it in a call-graph pruner of this type:

$$\begin{aligned}&\text{call-graph pruner} :\\ &\quad (\mathsf{CallGraph} \times \mathsf{Classifier} \times \mathsf{Threshold}) \rightarrow \mathsf{CallGraph}\end{aligned}$$

Algorithm 1 shows how a call-graph pruner works. Intuitively, a call-graph pruner uses a classifier to determine the probability that an edge in a static call graph is a true-positive. If that probability is below a given threshold $T \in$ Threshold, the call-graph pruner removes the edge.

Algorithm 1: Call-graph Pruner

```
1 Inputs: CallGraph $G$, Classifier $C$, Threshold $T$
2 let $G'$ be a copy of $G$
3 for every edge $e$ in $G$ do
4     $v$ = the feature values for $e$
5     if $C(v) < T$ then
6         remove $e$ from $G'$
7 Output $G'$
```

The threshold parameter enables us to explore different levels of aggressiveness in removing edges. For our example in Figure 2, we discussed a threshold of 50% in Section 2, which led to the removal of two edges. We could also use a lower threshold of 20%, which would lead to the removal of a single edge, namely the one from main to `A.foo`. The challenge is to strike a balance between removing many false-positive edges and keeping many true-positive edges. In

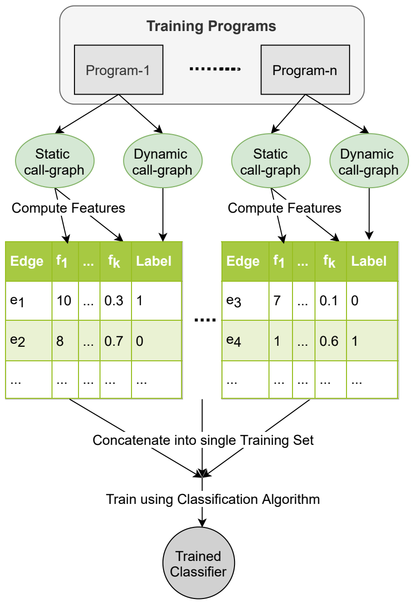


Figure 3: Classifier Generator workflow

Section 5 we will show results from an experimental investigation of how to choose a good threshold.

Notice that we use a static call graph constructor, a dynamic call graph constructor, and the training set of programs for the sole purpose of generating a classifier, while those items are no longer needed when we use the call-graph pruner.

3.2 Our Classifier Generator

We cast the edge-pruning problem as a classification problem for which learning a classifier can be done with machine learning. We proceed in three steps.

In the first step, we run existing static and dynamic call-graph constructor tools on every program in the training set (the dataset of programs is described in Section 4). The result is a set of pairs of call graphs: each pair consists of a static call graph and a dynamic call graph. We use the dynamic call graph as an approximation of the ground truth: if a static call-graph edge is also present in the dynamic call graph, we view it as a true-positive, and otherwise as a false-positive.

In the second step, for each program, we construct a table in which each row represents a static-call-graph edge. Figure 3 illustrates this table. The last column in each row (titled *Label* in Figure 3) contains a label of 1 or 0, based on whether the edge exists in the dynamic call graph. The remaining columns (titled f_1 to f_k) represent the set of *features* of the static call-graph edge. The example in Figure 2 uses two features: *dest-node-in-deg* and *src-node-out-deg*; we will discuss other features below. We can view each row in the table as a vector of feature-values. Concatenating the tables of each individual program gives us a single large training dataset of call-graph edges with ground truth labels. This training dataset consists of a large number of pairs (x_e, y_e), where x_e is a vector of feature values corresponding to a static call-graph edge, and y_e is a prediction of whether it is a false-positive or not. Our problem is now expressed in a format where it can be cast as a machine-learning classification problem [28].

In the third step we run an off-the-shelf machine-learning tool on the table constructed in second step. The result is a classifier that for any edge assigns a probability that it is a true-positive. We picked *random forests* [19] (ensembles of Decision Trees). One might try other approaches, which we leave to future work. Our goal with this step is to show that an off-the-shelf machine-learning tool is sufficient to get good results.

Our classifier generator can take any static call-graph constructor as input. For example, we have used the call-graph constructors WALA [47], Doop [9], and Petablox [33] as inputs and generated a call-graph pruner for each one.

The complexity of generating a classifier based on a training set with n edges is $O(n \log n)$ [19].

3.3 Our Feature set

Now we describe how we designed the feature set that both our classifier generator and our generated call-graph pruners use.

A *feature* is information about a static-call-graph edge that may help predict whether the edge is a true-positive. We would like our feature set to capture important context and semantic information about a call-graph edge. Encoding important semantic information as features is a common machine learning practice for incorporating domain knowledge into the learning process. For example, since dynamic dispatch is likely to affect the false-positive probability of a call-graph edge, we should add features that capture information about the targets of a method call. Using the context information of a graph edge has been useful for the related task of selective context and heap sensitivity in pointer-analysis [23], and we consider it a good criteria for picking features. Context information can be local by describing the neighborhood of the edge, or global by describing the call graph that the edge is a part of. In addition to capturing context and semantic features, we identify three criteria that we want our feature set to satisfy:

(1) linear-time computation complexity,
(2) interpretable and generalizable, and
(3) black-box.

The time-complexity guideline is particularly important given that some of our benchmarks can have several hundred thousand call-graph edges. Interpretability gives us an understanding of which call-graph edges are being dropped, and generalizability ensures that what is learned for the training edges also applies to call-graph edges of unseen programs. The black-box criterion implies that the features should only be designed on the output call graph, and not on some internal state or representation of a tool. This allows us to post-process the results without being specific to a particular algorithm or tool. Using these criteria, we arrived at the following features for an edge.

Figure 4 presents our feature set for an edge in a static call graph G, where the edge is from a caller method *caller* to a callee method *callee*. The node for the main method in G is *main*. The first seven features describe local information while the last four describe global information. Note that the L-fanout of an edge is the number of outgoing edges at the call-site of that particular edge, whereas

Feature	Description
src-node-in-deg	number of edges ending in *caller*
src-node-out-deg	number of edges out of *caller*
dest-node-in-deg	number of edges ending in *callee*
dest-node-out-deg	number of edges out of *callee*
depth	length of shortest path from *main* to *caller*
repeated-edges	number of edges from *caller* to *callee*
L-fanout	number of edges from the same call-site
node-count	number of nodes in G
edge-count	number of edges in G
avg-degree	average src-node-out-deg in G
avg-L-fanout	average L-fanout value in G

Figure 4: Our feature set

src-node-out-deg is the number of outgoing edges from all the call-sites of an entire source method.

Our selection process started with a much longer list of features that all satisfy the three criteria listed above. We picked from that list the ones that helped the most with removing false-positives. Our process used the training set as case studies to find the main reasons why tools give false positives. The result was the eleven features in Figure 4.

4 IMPLEMENTATION AND DATASET

Static Call-Graph Constructors. We used the static call-graph constructors WALA [47], Doop [9], and Petablox [33]. In each case we used the default setting, which implements 0-CFA for methods that are estimated to be reachable from the main method and without any special handling of reflection. Those tools produce significantly different call graphs and so we generate a separate call-graph pruner for each tool.

Reflection. In preliminary experiments, we found that enabling special handling of reflection in the static call-graph constructors introduces many false-positive edges in the call graphs. Our generated classifiers tend to assign each of those edges a low probability of being a true-positive, and therefore our call-graph pruners will correctly remove most of them. Therefore, special handling of reflection presents no additional challenge for call-graph pruning and we decided to go with the default setting of each static call-graph constructor.

Dynamic Call-Graph Constructor. We used the open-source tool Wiretap [26] to instrument the Java bytecode and thereby enable dynamic call-graph construction. Next, we ran the instrumented bytecode and collected data about the run, particularly about the method calls.

Standard Library. The Java standard library is large and has the potential to dominate the measurements for every benchmark, which is counterproductive. So, when we do our measurements and training, we omit nodes from the standard library as well as edges between standard library nodes. We preserve aspects of the edges to and from the standard library in the following way. For every path of the form

$$v \rightarrow \langle \ldots \text{ standard library nodes } \ldots \rangle \rightarrow w$$

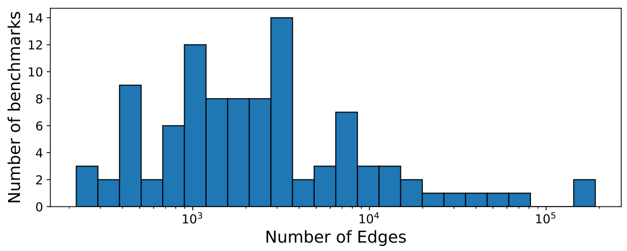


Figure 5: Histogram of Edge-counts in the 100 Training Programs.

where v, w are nodes outside the standard library, we create a single edge from v to w.

Random Forest Classifier. Our classifier generator uses the Random Forest algorithm [19] implemented with the Scikit-Learn [36] library (v0.21.3). The Random Forest algorithm works as follows: it trains several decision-trees using Bagging [10], and makes predictions by a "majority vote" across the decision trees. The training took 4 minutes. We tuned the hyper-parameters using Random Hyper-Parameter Search [5] with 4-fold cross-validation on the training set. We list the chosen hyper-parameters in the appendix.

Dataset. Our dataset consists of 141 programs from the NJR-1 benchmark suite [35], of which we used 100 programs for generating three call-graph pruners and the remaining 41 programs for our evaluation. We selected those 141 programs from the 293 NJR-1 programs according to the following criteria:

- consists at least 1,000 methods and at least 2,000 static call-graph edges according to Wala,
- executes at least 100 distinct methods at runtime, and
- has high coverage: executes a large percentage of the methods that are reachable from the main method according to Wala; for our benchmarks, the coverage is 68%, on average.

Each program consists of 560,000 lines of code, on average (not counting the standard library). In more detail, each program consists of the main application, which is 8,000 lines of code, on average, in addition to third-party libraries which account for an estimated 552,000 lines of code, on average.

The total number of static-call-graph edges (not counting the standard library) that are reachable from the main methods of the 141 programs is 1.3 million. For our classifier generator, each edge from 100 of those programs is a data point, which is 860,000 edges. Note that manual creation of ground truth about those 860,000 edges infeasible.

Large Benchmarks. The histogram in Figure 5 gives the distribution of the edge counts in the training programs. The X-axis is plotted on a logarithmic scale due to the skew in the distribution. Among the 100 training programs, 7 of them have a very large number of call-graph edges (> 20,000). This gives them the potential to dominate how the classifiers work. To overcome this, we randomly sample 20,000 edges from the edge-sets of these 7 programs. Notice that this sampling is done only during generation of call-graph pruners, while we use all the edges from the 41 programs that we use for evaluation.

Analysis Time. Running the three static call-graph constructors and the dynamic call-graph constructor on all the programs takes four days of compute time.

Precision and Recall. We estimate the quality of a static call graph using the standard notions of *precision* and *recall*. In our setting, if S is the edge set produced by a static call-graph constructor, and W is the edge set produced by Wiretap, then:

$$Precision = \frac{|S \cap W|}{|S|} \qquad Recall = \frac{|S \cap W|}{|W|}$$

The rate of false-positives is $(1 - Precision)$. We compute the average precision and recall values for the entire test-set by taking the arithmetic mean over the precision and recall values of individual programs.

Figure 6 shows a histogram of the original precision and recall scores for WALA on the 41 individual programs of the test set. Note that the precision values vary significantly, but almost all programs get below 40% precision. Hence, there is a lot of scope for improving the precision. The recall is close to 100% for most programs, but low for some due to heavy use of reflection, dynamic class-loading or native code.

5 EXPERIMENTAL RESULTS

In this section, we discuss our experimental results that validate the following claims.

(1) Our generated call-graph pruners for WALA, Doop, and Petablox produce call graphs with balanced 66% precision and 66% recall.
(2) For precision-sensitive clients, our generated call-graph pruners are significantly better at boosting precision than context-sensitive analyses, and have a much smaller overhead.
(3) The precision improvement is consistent across the test set.
(4) The call-graph pruner enables a monomorphic call-site client to balance its skewed 52% precision and 93% recall to a more balanced 68% precision and 68% recall.
(5) The call-graph pruner enables a null-pointer analysis to reduce its average warning count from 223 to 20, while increasing precision from 27% to 77%.

All experiments are run on a separate test set of 41 programs which were not used during training. The experiments were carried out on a machine with 24 Intel(R) Xeon(R) Silver 4116 CPU cores at 2.10GHz and 188 Gb RAM. A minimum RAM size of 32Gb is essential for ensuring that the static analyses run in reasonable time. The artifact for the paper is available here [46] and the NJR-1 dataset can be downloaded from [45].

5.1 Main Result

Figure 8 gives the main result of the paper: a call-graph pruner can be successfully used to boost precision and to balance the goals of precision and recall for the 0-CFA call-graph analysis of WALA, Doop and Petablox. The plot is used to represent the precision and recall values of various tools, wherein all precision and recall values are reported as averages over the test-set programs. The black triangle marks the WALA 0-CFA analysis (23.8% Precision, 95.3% Recall), the green triangle marks the Doop 0-CFA analysis (23.1% Precision, 92.6% Recall) and the blue triangle marks the Petablox 0-CFA analysis (29.8% Precision, 88.8% Recall). They all have close to perfect recall, but poor precision. The red plus sign marks the WALA 1-CFA analysis (29.6%. 95.4%). The black curve represents the precision-recall trade-off points obtained when a call-graph pruner is applied to the WALA 0-CFA output. The original WALA-0CFA output is a single point on the precision-recall graph, but the call-graph pruner gives a curve instead. This is because the call-graph pruner gives a probability score for each edge being in the ground-truth call-graph, and by setting different thresholds (i.e. cutoffs below which an edge is removed), we can obtain different points on the precision-recall curve. Joining all these different points gives us the black curve in the figure. Setting a low-probability threshold for accepting an edge, gives us points near the left end of the black curve, because we accept a large percentage of edges, thereby giving us higher recall but lower precision. Setting a high-probability threshold gives us points near the right end of the curve because we accept only very few edges which are very likely to be in the ground-truth call-graph, and this gives us high-precision and low recall. The green and blue curves represent the precision-recall trade-off obtained by applying the call-graph pruner to the Doop and Petablox call-graphs respectively, and the case is very similar to the black WALA curve.

These curves which trade-off recall for precision show that the classifier has assigned probabilities meaningfully. In contrast, a tool that randomly assigns probabilities to edges would result in a curve that goes straight down to zero recall without improving any precision. This is because it results in a random removal of edges, which keeps the ratio of true-positives (i.e. precision) the same. Boosting precision requires the ratio of false-positive edges in the removed edge set to be higher than the rest of the edges.

There are 2 particularly interesting points on the black (WALA) curve in Figure 8. The first is the one marked by the black (WALA) square (66.0% Precision, 66.0% Recall), which represents the point with balanced precision and recall. Such a point will be useful to a precision-sensitive client analysis. As compared to the original WALA 0-CFA (black-triangle), this point has over 72% of the edges from the original call-graph removed, and out of the removed edges, less than 10% are true positives. This point is at a 0.45 probability threshold. Similar points for Doop and Petablox, marked by a green square (hidden behind the black square) and blue square (also hidden behind the black square) respectively, are at (66.2% Precision, 66.2% Recall) and (66.4% Precision, 66.4% Recall) respectively. A second interesting point is the right-most point on the curve after which recall starts dropping faster, represented by a black circle (50% Precision, 92% Recall). Such a point would be useful for a client analysis that needs to increase a little precision, without losing much recall. Similar points for Doop and Petablox are marked by the green circle (50% Precision, 88% Recall) and blue circle (50% Precision, 87% Recall) respectively.

Both these points give larger precision boosts than the 1-CFA analysis. However, in general, the best precision-recall trade-off point is decided by the needs of the client of the call graph. Precision-sensitive clients would benefit more from our call-graph pruner

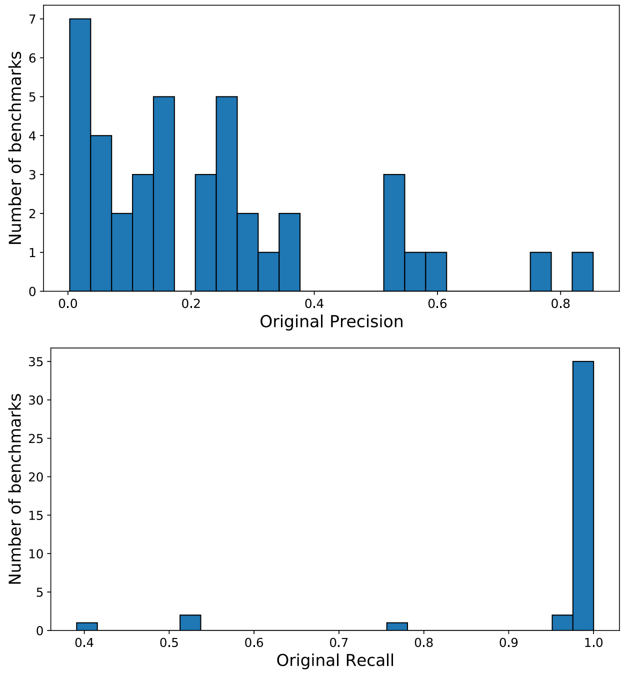


Figure 6: Precision and recall for 41 test programs.

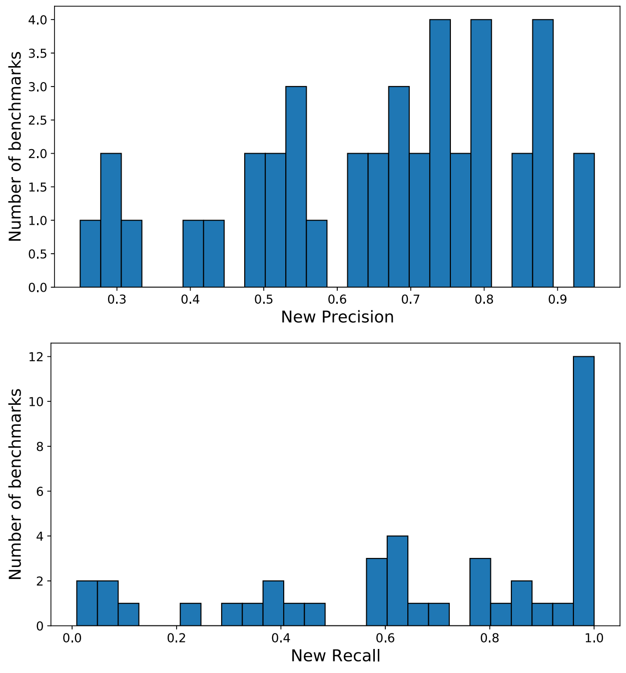


Figure 7: Precision and recall after call-graph pruning.

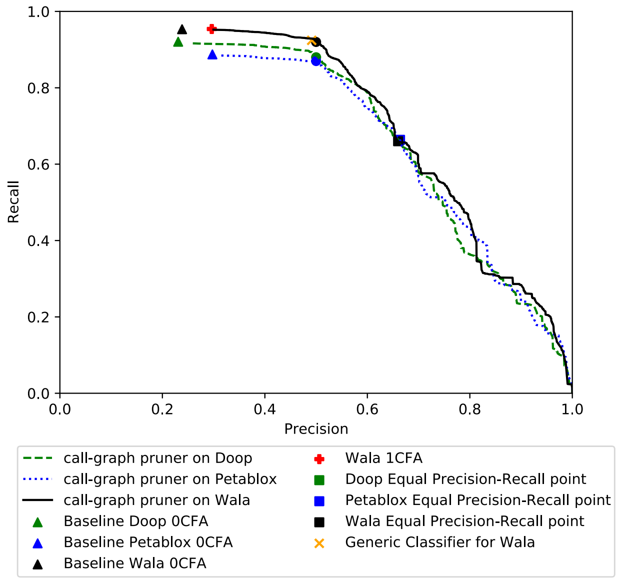


Figure 8: Main Result for the WALA, Doop and Petablox static analysis tools. The baseline precision-recall values for the 3 tools, along with the precision-recall curve obtained after applying a call-graph pruner (averaged over all test programs)

since it gives a larger precision boost, but clients that need high recall may prefer the 1-CFA call graph.

Our call-graph pruner adds an overhead of 18% to the WALA 0-CFA analysis, whereas moving to a 1-CFA analysis adds 292% overhead. Prior research also finds that context-sensitivity increases analysis time by many folds [30].

For completeness, we also ran this experiment for WALA's RTA implementation and it gets similar results (that we show in the supplementary material). Since the three tools show similar characteristics, we only present numbers for the WALA 0-CFA call graph in the rest of this section. The corresponding graphs for Doop and Petablox are available in the supplementary material.

Picking a Cutoff value. We picked the balanced precision-recall point because it gave good results for a null-pointer analysis client, but different precision-recall trade-off points may be suitable for different client analyses. Figure 9 helps a user pick the right trade-off point for their client. It plots the probability cutoff values on the X-axis, and the Precision, Recall and F-score on the Y-axis. The graph shows what values each of these metrics takes at every probability cutoff value, as well as what the expected cutoff would be for a given target Precision, Recall or F-score. For example, by looking at the figure, we can say that to obtain an expected Precision of 60%, we can set a cutoff value of 0.4. At this point we would get a Recall of approximately 75% and F-score of around 65%. This graph also shows that the balanced precision-recall point is also very close to the point with maximum F-score.

Feature Importance. Figure 10 gives the impurity-based importance [42] for each feature used in the random-forest in descending order. The *L-fanout* and *dest-node-in-deg* are the most important features and the four global features are the least important. Dropping

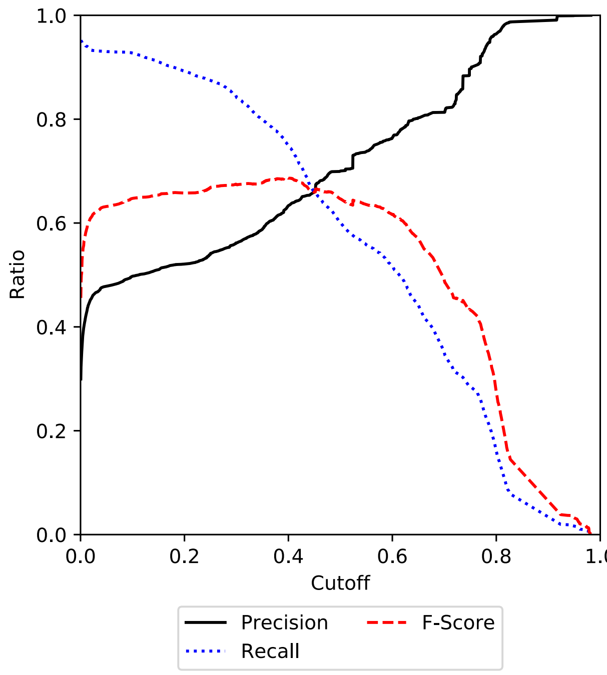


Figure 9: Probability cutoff plotted vs Precision, Recall and F-score curves for WALA

Feature	Importance
L-fanout	0.182
dest-node-in-deg	0.114
src-node-out-deg	0.094
repeated-edges	0.092
src-node-out-deg	0.090
depth	0.084
dest-node-out-deg	0.079
node-count	0.071
edge-count	0.067
avg-L-fanout	0.036
avg-degree	0.028

Figure 10: Importance of each feature in the Random Forest Classifier in descending order.

the four global features decreases the area under the precision-recall curve from Figure 8 by 6%.

Human-Interpretable Explanation of the Classifiers. We can give a human-interpretable explanation of the main aspects of the Random Forest classifiers that were learned in the experiment. In each case, the top-level decisions center around the following generic classifier:

$$\text{if ((L-fanout} > m) \wedge \text{(dest-node-in-deg} > n\text{)) then 0 else 1}$$

The above expression says that if an edge has *L-fanout* greater than m and *destination-node in-degree* greater than n, then the probability that it is a true edge is 0, and otherwise 1. For each of the static call-graph constructors, we can identify the constants m and n:

WALA:
if ((L-fanout > 3.5) $\wedge$ (dest-node-in-deg > 9.5)) then 0 else 1
Doop:
if ((L-fanout > 3.5) $\wedge$ (dest-node-in-deg > 16.5)) then 0 else 1
Petablox:
if ((L-fanout > 3.5) $\wedge$ (dest-node-in-deg > 20.5)) then 0 else 1

The orange cross (49% precision, 92% recall) in Figure 8 gives the precision-recall trade-off when using the generic classifier for WALA. This generic classifier has a slightly worse trade-off and is much less tunable than the black line (WALA with call-graph pruner). However, its pruning rules are also much simpler and easily understandable. The use of *L-fanout* and *dest-node-in-deg* in the generic classifier aligns with the fact that these are the most important features according to Figure 10.

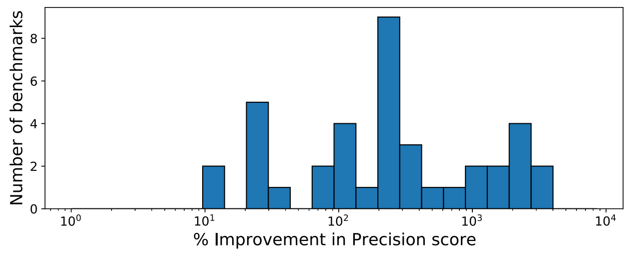


Figure 11: Historgram of Percentage Improvement in Precision scores for individual programs.

5.2 Distribution of Precision and Recall for individual programs

Figure 7 gives a histogram of the precision and recall scores of individual programs when a call-graph pruner is used to prune the WALA call graph at the balanced precision-recall point (marked by the black square in Figure 8). Most of the programs get at least 50% precision, and a several even reach the 70% precision goal. Contrast this to the precision in Figure 6 where almost all programs fail to cross the 40% precision point.

As expected, the recall scores from Figure 7 dropped as compared to Figure 6. However, most programs still get at least 50% recall, implying that they retain a good portion of their true edges. Note that it is impossible to improve recall using a call-graph pruner since it cannot find new edges that WALA did not find.

The histogram from Figure 11 illustrates the percentage improvement in precision scores. The X-axis is plotted on a logarithmic scale. By using a call-graph pruner, 30 out of the 41 programs have their precision score boosted by at least 2 times their original precision score. All but 2 programs have their precision score boosted by at least 20%. No benchmark gets a worse precision. Thus, a significant majority of the individual programs consistently get a large precision improvement without loosing too much recall, and achieve a better precision-recall balance. The Doop and Petablox graphs have similar characteristics and are shown in the supplementary material.

Call-graph tool	Precision	Recall
WALA 0-CFA	51.8%	92.6%
WALA 0-CFA + call-graph pruner	67.7%	68.4%

Figure 12: Impact of improved call-graph precision on a monomorphic call-sites client

ID	Warnings		True-Positives in a sample of 10	
	Before	After	Before	After
B1	137	12	2	10
B2	365	31	4	5
B3	190	15	2	8
B4	308	44	7	10
B5	204	16	0	10
B6	429	42	0	7
B7	404	136	7	10
B8	70	10	0	0
B9	231	10	0	9
B10	102	34	5	8
Average			2.7	7.7

Figure 13: Total warning counts and a manual classification of a sample of 10 warnings for the null-pointer analysis before and after applying a call-graph pruner

5.3 Effect on Client Analyses

Next, we look at the effect of improved call-graph precision on the monomorphic call-site detection and null-pointer analysis clients.

Monomorphic call-site client. This client is based on the WALA-generated 0-CFA call graph, and it uses the dynamic analysis as the ground-truth. Figure 12 give the precision and recall of a monomorphic call-site client with and without the call-graph pruner. The call-graph pruner helps the client boost precision from 52% to 68% and balance its goals of precision and recall.

Applications of the monomorphic call-sites client include devirtualization and inlining. Since the call-graph analysis is never sound in practice [31], these applications require some safety checks, resulting in overheads. For example, if devirtualization is used for optimization, run-time checks need to be inserted to ensure correctness [22]. Higher precision for the monomorphic call-sites client implies that more of the call-sites declared monomorphic by the static analysis actually turn out monomorphic in the ground-truth. This in turn implies that whenever we incur the overhead of inlining or devirtualization, we are also more likely to realize its benefits.

Null pointer analysis. This analysis is based on the paper by Hubert et al. [21]. It is implemented in WALA, and is used to find null-pointer errors originating from uninitialized instance fields. The analysis is context-insensitive, field-insensitive and flow-sensitive. It only reports potential null-pointer dereferences in application code, and not for the standard library.

The original WALA call graph gives us, on average, 223 null pointer warnings per program. The high volume of warnings makes it cumbersome for developers to manually inspect and in practice this results in developers ignoring the tool output entirely [6, 25]. Using the call graphs produced after pruning gives us much fewer (on average 20 per program) warnings.

Two of the authors manually inspected a random sample of 10 null-pointer warnings from 10 of the 41 test programs when used with and without the call-graph pruner. The 10 programs were chosen with the criteria that they had at least 10 warnings both with and without the call-graph pruner, and the ratio of warnings with and without the call-graph pruner was close to (20/223). Figure 13 gives the total warning counts as well as the true-positive counts (from a sample of 10 warnings) for each of these 10 programs. The use of a call-graph pruner helped the null-pointer analysis improve its precision from 27% to 77%

The criteria for marking a warning as a true-positive was that the author could trace the backward slice of a dereference to an instance field which was uninitialized by the end of a constructor. Warnings that either could not be verified in 10 minutes, ran into another exception before triggering the null exception, or otherwise unverifiable by the authors, were considered as false-positives. Reachability from the main method was not considered because it is hard to verify manually.

We leave to future work to try other clients, including other approaches to null-pointer analysis such as NullAway [4].

5.4 Threats to Validity

The first threat is the use of a dynamic analysis as a proxy for the call-graph ground truth. It assumes good coverage of the true ground-truth call-graph and affects the precision-recall calculations. If the dynamic analysis had higher coverage, more of the static analysis edges would be in the dynamic call-graph. As a consequence, both the baseline precision scores as well as the pruned-call-graph precision scores would be higher. In contrast, we expect the recall scores to remain similar. However, improving dynamic analysis coverage is a non-trivial and orthogonal problem and any techniques improving coverage will automatically improve our technique and evaluation. Symbolic execution [27] is one option to improve coverage, but it doesn't scale to the size of our programs. Instead, we use a subset of the NJR-1 benchmark set which gets good coverage. Note that this threat does not affect the evaluation of the null-pointer analysis.

The second threat is the manual inspection of the null-pointer warnings, which are vulnerable to human errors. The authors inspecting the errors have a limited familiarity with the code-bases of the examined program. This could lead to misclassification of both true and false errors, and affect the precision score accordingly. Further, the precision scores are reported for a sample of 10 programs.

The third threat to validity is the generalizability of the results to programs outside the NJR dataset. Our assumption is that our learning and evaluation results generalize to other programs outside the dataset.

The fourth threat to validity is that programs in the training set and evaluation set share some third-party libraries. On average (geometric mean), 3.6 percent of the methods of a program in the evaluation set also occur in some training program. We believe that this overlap is low enough to not significantly affect the conclusions of our evaluation.

6 RELATED WORK

Our technique is the first to apply machine learning to boost call-graph precision. In our discussion of related work, we focus on three areas: combining static and dynamic analyses, applying machine learning to remove static-analysis false-positives, and improving the precision of call-graph construction.

Combining static and dynamic analysis. Prior research has used a dynamic analysis to improve the precision of a static analysis. Grech et. al [16] generate dynamic heap information and use this as a drop-in replacement for the heap modeling part in an existing static analysis tool to improve its precision. Artzi et. al [3] use a dynamic analysis to confirm the mutability information computed by a static analysis. Chen et. al [13] use the information from test-executions to prioritize the alarms given by a static analysis. The main drawback that these tools face is that they need the dynamic analysis to be run every single time the tool is run. In contrast, our technique needs the dynamic analysis only for generating a call-graph pruner. After that, a call-graph pruner is purely a static tool, and hence does not suffer from the usual drawbacks of a dynamic analysis like long execution times or finding good inputs.

Applying machine learning to improve static-analysis by removing false-positives. The technique of filtering static-analysis false-positives by casting it to a classification problem with hand-picked features has been used for static bug-analysis tools [15, 18, 39, 44, 49]. Each of these works follows the same workflow: collect static analysis error-reports, get a programmer to label them as true or false-positives, design a feature-set for the error reports, and then train a classifier on these labeled error-reports to identify false-positives. However, they have minor differences among themselves in terms of the feature-set chosen, the bug-reporting tool used and the benchmarks used for the training data. Ruthruff et. al [39] use the FindBugs [20] bug-reporting tool and the set of Java programs at Google as their dataset. Heckman and Williams [18] also use FindBugs reported bugs on 2 open-source Java projects. Yuksel and Sozer [49] classify bug-alerts for a digital TV software. Flynn et al. [15] combine the bug-alerts from multiple tools, in addition to using the hand-picked features. Tripp et. al [44] work with a JavaScript security checker's warnings from popular Web sites as its dataset.

Our work differs in three ways: it uses an estimate of ground-truth produced by dynamic analysis, it has a generalizable approach to picking a feature set, and it has a tunable precision-recall trade-off, as we discuss next.

The key bottleneck faced by each of these prior works was that they relied on the collection of human-labeled ground-truth, which does not scale. This restricted their dataset to a handful of projects and a couple of thousand data-points (bug reports) at best. In fact, for each type of error, there is typically less than a few hundred bugs in each of the datasets. In contrast, our technique uses an estimate of ground-truth produced by dynamic analysis, which allows it to scale to a much larger number of programs with a million data points (call-graph edges).

The second major difference is in the choice of the feature-set. This is partly a consequence of the fact that the previous work focuses on static-analysis error report data, which is different from the graph output generated by call-graph construction tools. Hence some of the common features used in these works are the bug-priority level, file-modification-frequency, coding-style metrics, and lexical features (like method or package names). These features, though appropriate, violate generalizability and black-box guiding principles listed in Section 3.3. Non-black-box features like bug-priority level will not generalize across different tools or algorithms, and non-generalizable features like lexical features are unlikely to generalize to programs outside the dataset. In contrast, we use a systematic approach to selecting features, as described in Section 3.3, and as a consequence, our approach generalizes easily across multiple programs and multiple call-graph construction tools.

The third difference is that these prior works, except for [44], provide a single precision-recall point. [44] provide eight different precision-recall points, by varying the classifier used. Instead, our approach has a tunable precision-recall trade-off by predicting edge-probabilities and pruning edges with probability lower than a threshold. Further, we only use a single classifier (Random Forests) since it achieves superior precision-recall trade-offs than the classifiers used in [44].

Another area that uses machine learning for filtering false positive is the work by Raghothaman et al. [37]. They predict the probabilities of static-analysis alarms using Bayesian inference and update these as the user resolves alarms as true or false positives. This paradigm of online learning, where the model is learned and improved as the user gives feedback, is quite different from our fully-automated offline learning paradigm, where we do a one-time training on a large dataset of static and dynamic analysis outputs and require no user input.

Recently data-driven techniques have also been used to selectively apply context- and flow-sensitivity [12, 24] to methods that will benefit it the most. These techniques can potentially provide the precision improvement of a 1-CFA at a lower overhead, but as seen in Figure 8, this improvement is still much lower than what is achieved by our call-graph pruner.

Improving the precision of call-graph construction. Lhotak [29] designed an interactive tool to qualitatively understand the root cause of differences between different static and dynamic analysis tools. This is then used in a case study to understand the main cause of imprecision in a static analysis tool as compared to its corresponding dynamic analysis output. In contrast, our classifier generator is fully automated, using machine learning, and doesn't require a skilled programmer to use an interactive tool to figure out the cause of the imprecision.

Sawin and Rountev [41] propose certain heuristics to deal with dynamic features like reflection, dynamic class loading and native method-calls in Java, which helps to improve call-graph precision of the CHA algorithm without sacrificing much recall. Similarly, a call-graph pruner trades of a little recall for a large boost in precision, but it achieves this through automated machine learning on a dataset of call graphs instead, and is able to boost precision by a much larger amount. Additionally, we work with a 0-CFA baseline (with no handling of dynamic features like reflection), which already has a large precision gain over a CHA algorithm with reflection handling.

Zhang and Ryder [50] create precise application-only call graphs by identifying which edges from the standard library to the application are really false-positive. This is similar to the precision boost we gain for the edges that go via the standard library. However, we generate a classifier that learns this on its own from data, and we use the classifier in a call-graph pruner that is able to boost precision even further.

The patent by Reif et. al [32] uses probabilities to quantify analysis imprecision. Each analysis constraint is assigned a probability heuristically or via user configuration, and the probabilities for call-graph edges are derived from these using a type-propagation graph. In contrast, our call-graph pruner learns all its edge probabilities from data about static and dynamic call-graphs. Further, while their technique calls for a new static analysis, our call-graph pruner works as a black-box post-processor for existing call-graph construction tools.

More distantly related is the work by Blackshear et. al [8], which prunes control-flow edges representing interleavings between events in an event-driven system. This pruning task is different from our task which focuses on pruning call-graphs edges for sequential code.

There has also been prior work that uses a dynamic analysis to evaluate call-graph related static analysis tools [1, 11, 16, 38, 43]. Our tool additionally uses the dynamic analysis results as training labels to prune the result from a static call-graph construction tool.

7 CONCLUSION AND FUTURE WORK

Our approach to generating a high-precision call graph first runs an existing black-box call-graph constructor and then prunes the resulting call graph. A call-graph pruner uses a classifier, which is trained on a large number of static and dynamic call graphs, to predict the probability of an edge being a true-positive. Using different thresholds for the edge probabilities we can tune the precision-recall trade-off of the call graph. We empirically showed how a call-graph pruner can be used to boost precision and balance the recall and precision of call graphs produced by WALA, Doop and Petablox, which are otherwise skewed towards high recall and low precision. We also ran a null-pointer analysis and a monomorphic call-sites analysis with these pruned call graphs, and we showed that they got much closer to the high-precision expectations of their users.

Future work includes automatically learning a feature-set for use by our pruner generator and our generated call-graph pruners. A particularly promising avenue for future work is to explore graph neural networks for automatic feature-learning. Recent work has used graph neural networks [17] for program analysis tasks like program similarity [34], variable misuse prediction [2, 48] variable name prediction [2], and method name prediction [48]. The features that are discovered in those papers are not features of call graphs and hence this remains an open problem.

A second future direction could be to replace dynamic-analysis ground-truth labeling with developer-labeling for call-graph edges. The challenge here is that the cumulative number of edges in the training dataset is nearly a million, and developer-labeling doesn't scale to such a large dataset.

A third future direction could be to adapt our technique to heap-reachability queries [7].

ACKNOWLEDGMENTS

This work was supported by the U.S. NSF Award 1823360, and the ONR Award N00014-18-1-2037. We thank Aishwarya Sivaraman, Aaron Berdy and Ashwin Dharne for many helpful discussions. We also thank the ICSE'22 reviewers for their constructive comments that helped improve the paper.

APPENDIX

The example in Figure 2 is an excerpt of from the program that Figure 14 shows in full.

Our classifier generator uses the Random Forest algorithm [19] implemented with the Scikit-Learn [36] library (v0.21.3). We tuned the hyper-parameters using Random Hyper-Parameter Search [5]. The score for which we optimized was the *area under the precision-recall curve* and Figure 15 lists the chosen hyper-parameters.

```
class A{                 public class Main{
    A f;                     static A id(A a){
    void foo(A a){               new A().foo(a);
        a.toString();            return a;
    }                        }
}                            static A getObjC(){
                                 new A().foo(new A());
class B extends A{               new B().foo(new A());
    void foo(A b){               A p = id(new A());
        b.toString();            A q = id(new B());
    }                            A r = id(new C());
}                                return r;
                             }
class C extends B{           public static void main(
    void foo(A c){                   String[] args){
        c.toString();            A x = getObjC();
    }                            x.foo(x.f);
}                                x.f = new A();
                             }
                         }
```

Figure 14: Program for the example in Section 2

Hyperparameter	Value
Number of Trees	1000
Maximum Depth	10
Bootstrapping	False
Minimum samples for split	2
Maximum features for split	sqrt(feature count)
Minimum samples for leaf	1
Split quality criterion	Entropy
Other hyper-parameters	Library default

Figure 15: Hyper-parameters for Random-Forests

REFERENCES

[1] Karim Ali and Ondřej Lhoták. 2012. Application-Only Call Graph Construction. In *ECOOP 2012 – Object-Oriented Programming*, James Noble (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 688–712.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=BJOFETxR-

[3] Shay Artzi, Adam Kiezun, David Glasser, and Michael D. Ernst. 2007. Combined Static and Dynamic Mutability Analysis. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA) *(ASE '07)*. Association for Computing Machinery, New York, NY, USA, 104–113. https://doi.org/10.1145/1321631.1321649

[4] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. NullAway: Practical Type-Based Null Safety for Java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 740–750. https://doi.org/10.1145/3338906.3338919

[5] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-parameter Optimization. *J. Mach. Learn. Res.* 13, 1 (Feb. 2012), 281–305. http://dl.acm.org/citation.cfm?id=2503308.2188395

[6] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[7] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2013. Thresher: Precise Refutations for Heap Reachability. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 275–286. https://doi.org/10.1145/2491956.2462186

[8] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2015. Selective Control-Flow Abstraction via Jumping. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 163–182. https://doi.org/10.1145/2814270.2814293

[9] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. ACM, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[10] Leo Breiman. 1996. Bagging predictors. *Machine Learning* 24, 2 (01 Aug 1996), 123–140. https://doi.org/10.1007/BF00058655

[11] Raymond P. L. Buse and Westley Weimer. 2009. The Road Not Taken: Estimating Path Execution Frequency Statically. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 144–154. https://doi.org/10.1109/ICSE.2009.5070516

[12] Sooyoung Cha, Sehun Jeong, and Hakjoo Oh. 2018. A scalable learning algorithm for data-driven program analysis. *Information and Software Technology* 104 (2018), 1–13. https://doi.org/10.1016/j.infsof.2018.07.002

[13] Tianyi Chen, Kihong Heo, and Mukund Raghothaman. 2021. Boosting Static Analysis Accuracy with Instrumented Test Executions. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1154–1165. https://doi.org/10.1145/3468264.3468626

[14] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 332–343. https://doi.org/10.1145/2970276.2970347

[15] Lori Flynn, William Snavely, David Svoboda, Nathan VanHoudnos, Richard Qin, Jennifer Burns, David Zubrow, Robert Stoddard, and Guillermo Marce-Santurio. 2018. Prioritizing Alerts from Multiple Static Analysis Tools, Using Classification Models. In *Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies* (Gothenburg, Sweden) *(SQUADE '18)*. Association for Computing Machinery, New York, NY, USA, 13–20. https://doi.org/10.1145/3194095.3194100

[16] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2018. Shooting from the Heap: Ultra-Scalable Static Analysis with Heap Snapshots. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 198–208. https://doi.org/10.1145/3213846.3213860

[17] Aditya Grover and Jure Leskovec. 2016. Node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 855–864. https://doi.org/10.1145/2939672.2939754

[18] Sarah Heckman and Laurie Williams. 2009. A Model Building Process for Identifying Actionable Static Analysis Alerts. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation (ICST '09)*. IEEE Computer Society, USA, 161–170. https://doi.org/10.1109/ICST.2009.45

[19] Tin Kam Ho. 1995. Random Decision Forests. In *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1 (ICDAR '95)*. IEEE Computer Society, USA, 278.

[20] David Hovemeyer and William Pugh. 2004. Finding Bugs is Easy. *SIGPLAN Not.* 39, 12 (Dec. 2004), 92–106. https://doi.org/10.1145/1052883.1052895

[21] Laurent Hubert, Thomas Jensen, and David Pichardie. 2008. Semantic Foundations and Inference of Non-null Annotations. In *Formal Methods for Open Object-Based Distributed Systems*, Gilles Barthe and Frank S. de Boer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 132–149.

[22] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. 2000. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Minneapolis, Minnesota, USA) *(OOPSLA '00)*. Association for Computing Machinery, New York, NY, USA, 294–310. https://doi.org/10.1145/353171.353191

[23] Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning Graph-Based Heuristics for Pointer Analysis without Handcrafting Application-Specific Features. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 179 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428247

[24] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-Driven Context-Sensitivity for Points-to Analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 100 (oct 2017), 28 pages. https://doi.org/10.1145/3133924

[25] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 672–681.

[26] Christian Gram Kalhauge and Jens Palsberg. 2018. Sound Deadlock Prediction. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 146 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276516

[27] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Warsaw, Poland) *(TACAS'03)*. Springer-Verlag, Berlin, Heidelberg, 553–568.

[28] S. B. Kotsiantis. 2007. Supervised Machine Learning: A Review of Classification Techniques. In *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering*. IOS Press, NLD, 3–24.

[29] Ondrej Lhoták. 2007. Comparing Call Graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (San Diego, California, USA) *(PASTE '07)*. Association for Computing Machinery, New York, NY, USA, 37–42. https://doi.org/10.1145/1251535.1251542

[30] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-First Pointer Analysis with Self-Tuning Context-Sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/3236024.3236041

[31] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. https://doi.org/10.1145/2644805

[32] Yi Lu, Daniel Wainwright, and Michael Reif. 2020. Probabilistic call-graph construction. (Jul 2020). US Patent No. 10,719,314 B2.

[33] Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. 2015. A User-Guided Approach to Program Analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 462–473. https://doi.org/10.1145/2786805.2786851

[34] Aravind Nair, Avijit Roy, and Karl Meinke. 2020. FuncGNN: A Graph Neural Network Approach to Program Similarity. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (Bari, Italy) *(ESEM '20)*. Association for Computing Machinery, New York, NY, USA, Article 10, 11 pages. https://doi.org/10.1145/3382494.3410675

[35] Jens Palsberg and Cristina V. Lopes. 2018. NJR: A Normalized Java Resource. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops* (Amsterdam, Netherlands) *(ISSTA '18)*. Association for Computing Machinery, New York, NY, USA, 100–106. https://doi.org/10.1145/3236454.3236501

[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python . *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[37] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-Guided Program Reasoning Using Bayesian Inference. *SIGPLAN Not.* 53, 4 (June 2018), 722–735. https://doi.org/10.1145/3296979.3192417

[38] Atanas Rountev, Scott Kagan, and Michael Gibas. 2004. Static and Dynamic Analysis of Call Chains in Java. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Boston, Massachusetts, USA) *(ISSTA '04)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/1007512.1007514

[39] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. 2008. Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 341–350. https://doi.org/10.1145/1368088.1368135

[40] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66. https://doi.org/10.1145/3188720

[41] J. Sawin and A. Rountev. 2011. Assumption Hierarchy for a CHA Call Graph Construction Algorithm. In *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*. 35–44.

[42] Scikit-learn. [n.d.]. Feature importances with a forest of trees. https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html.

[43] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. 2020. On the Recall of Static Call Graph Construction in Practice *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1049–1060. https://doi.org/10.1145/3377811.3380441

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

[45] Akshay Utture, Christian Gram Kalhauge, Shuyang Liu, and Jens Palsberg. 2020. *NJR-1 Dataset*. https://doi.org/10.5281/zenodo.4839913

[46] Akshay Utture, Christian Gram Kalhauge, Shuyang Liu, and Jens Palsberg. 2021. *Artifact for ICSE-22 submission "Striking a Balance: Pruning False-Positives from Static Call Graphs"*. https://doi.org/10.5281/zenodo.5177161

[47] WALA. 2015. IBM, "T.J. Watson Libraries for Analysis (WALA),". http://wala.sourceforge.net.

[48] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning Semantic Program Embeddings with Graph Interval Neural Network. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 137 (Nov. 2020), 27 pages. https://doi.org/10.1145/3428205

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

[50] Weilei Zhang and Barbara G. Ryder. 2007. Automatic Construction of Accurate Application Call Graph with Library Call Abstraction for Java: Research Articles. *J. Softw. Maint. Evol.* 19, 4 (July 2007), 231–252.