Is Call Graph Pruning Really Effective?

An Empirical Re-evaluation

Mohammad Rafieian
The University of Texas at Dallas
Richardson, Texas, USA
mohammad.rafieian@utdallas.edu

Vlad Birsan
The University of Texas at Dallas
Richardson, Texas, USA
vlad.birsan@utdallas.edu

Kunal Katiyar
Coppell High School
Coppell, Texas, USA
kxk6886@g.coppellisd.com

Dylan Zhong
Westlake High School
Austin, Texas, USA
hideyoshi5823@gmail.com

Shiyi Wei
University of Texas at Dallas
Richardson, Texas, USA
swei@utdallas.edu

Abstract

Recently proposed call graph pruning techniques, which use machine learning to predict and reduce false positives in static call graphs, have reported impressive results and demonstrated practicality for downstream analyses. However, as we research the methodology of how datasets are created and how evaluation is conducted in these research projects, we identify many risk factors that may make the reported results incomplete or even invalid. We further investigate these factors and provide empirical evidence to demonstrate they indeed result in misleading results. Motivated by these findings, we propose an empirical re-evaluation of existing call graph pruning techniques to assess their effectiveness. We first construct a new dataset of real-world programs, applying three complementary labeling approaches, resulting in 41,952 labels. To provide a full picture of these techniques, we utilize three popular call graph analysis frameworks (WALA, Doop, and OPAL) and seven tool configurations. Our results show that pruning is not as effective as reported in prior work, improving precision at the cost of significant coverage loss. We further train more general models across analysis tools and configurations, and explore the impact of data splitting and balancing. We find these models often match or outperform the existing configuration-specific ones, indicating their applicability in more general settings. Finally, we highlight the open challenges and potential solutions, including the need for better datasets as well as improved feature engineering.

CCS Concepts

• **Software and its engineering** → **Automated static analysis**; • **Theory of computation** → *Program analysis*; • **Computing methodologies** → Machine learning.

Keywords

Call Graph, Static Analysis, Machine Learning, Empirical Study

ACM Reference Format:
Mohammad Rafieian, Vlad Birsan, Kunal Katiyar, Dylan Zhong, and Shiyi Wei. 2026. Is Call Graph Pruning Really Effective?: An Empirical Re-evaluation. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773247

1 Introduction

Call graph analysis, which approximates calling relationships in a program, is a fundamental component of many static analysis tasks, including optimization and vulnerability detection. Over the past decades, numerous algorithms have been proposed [1, 37, 45, 51], each offering different tradeoffs between soundness, precision, and performance. These tradeoffs heavily influence their practicality in real-world settings. For example, Smaragdakis et al. [52] showed that precise context-sensitive analyses with $k > 2$ do not scale to large programs. As a result, developers often resort to less precise analyses for scalability.

A persistent challenge of static call graph analysis is the high rate of false positives (i.e., edges that cannot occur at runtime) introduced by imprecise call graph construction algorithms. Other sources of imprecision, such as calls through arrays [47], further degrade call graph accuracy. Excessive false positives remain a major barrier to adopting static analysis tools because they impose substantial manual-inspection costs [56]. Improving call graph precision can therefore reduce downstream false alarms and enhance the overall usability of static analysis tools.

These limitations motivate the use of machine learning to prune false edges from highly sound but imprecise call graphs, potentially achieving precision comparable to—or even surpassing—context-sensitive analyses with $k > 2$. Prior work [56] demonstrated that ML-based pruning can significantly improve precision (e.g., +40%), and subsequent studies [31, 38] further advanced this direction.

For example, cgPruner [56] uses structured features with random forest model, while AutoPruner [31] uses a fine-tuned codeBert [12] model to extract semantic features and a neural classifier to make the prediction.

These ML-based call graph pruning approaches need to be trained on labeled datasets and evaluated empirically. In all existing papers [31, 38, 56], the datasets (e.g., NJR1 [44] and NYXCorpus [38]) are created following the same methodology: using the dynamic call graphs as the ground-truth true edges. Specifically, these approaches are trained treating the dynamic call graph edges as true edges, while *any edges that do not appear in the dynamic call graph are assumed as false*. However, because a dynamic call graph *under-approximates* an actual call graph, important questions on the existing call graph pruning approaches shall be answered: *are these datasets suitable for training the call graph pruning models and are the existing evaluations trustworthy?*

To answer these questions, we randomly sampled and manually labeled 348 edges that are assumed to be false in the NJR1 dataset, and found that 313 out of the 348 edges are actually true (Section 2). Training the existing approaches on only known true and false edges yields significantly different results (no precision improvement) compared to assuming dynamic call graphs as ground truth (13% precision improvement). We additionally observe that the programs in NJR1 may not contain complex invocations, and the evaluated call graphs ignore reflection handling, a well-known source of over-approximation in call graph analysis [47, 49, 53]. These observations lead us to believe that the existing datasets are not suitable for the call graph pruning task and the existing evaluation results may be incomplete and even misleading.

Motivated by the above findings, our work aims to re-evaluate the call graph pruning techniques to understand the state of the art. To achieve this goal, we first create a dataset that satisfies two criteria: (1) consisting of real-world programs, and (2) containing both true and false validated call graph edges. Specifically, our new dataset is created from 4 real-world programs, derived from the XCorpus benchmark [15]. The labels are generated using three complementary approaches. First, we adopt the true labels generated by Helm et al. [15] in these 4 programs, using a semi-automated dynamic fuzzing-based approach. Second, we adopt the metamorphic relationship between analysis options, first presented by Mordahl et al. [41], in three call graph analysis frameworks (WALA [19], Doop [5], and OPAL [16]) to automatically label the false edges. Third, for the remaining unlabeled edges, we randomly sample and manually label them to produce true and false validated labels. In total, our dataset includes 41,952 labeled call graph edges.

Using this new dataset, we re-evaluate the call graph pruning techniques through the lens of various configurations in WALA, Doop, and OPAL. We utilize five machine learning models, three from prior work that use random forest, and fine-tuned codeBERT and codeT5+ [31, 38, 56], and two new variants using pre-trained codeBERT and codeT5+. First, we replicate the setup of existing evaluations, training a separate model for each tool configuration. Contrary to prior findings, we observe that these ML-based pruning techniques do not perform effectively, and the model features are insufficient to reliably distinguish between false and true edges. Second, we train more general models, on all labels per tool using all its configurations, and even a single general model across all call graphs. Interestingly, we find that these new models perform comparably to, or even better than, models trained on a specific configuration, suggesting the practical utility of the general models. Our evaluation also investigates the impact of different data balancing and splitting techniques. Finally, we discuss limitations of current pruning techniques, including the lack of representative ground-truth data and inadequate feature representations, and identify promising future directions.

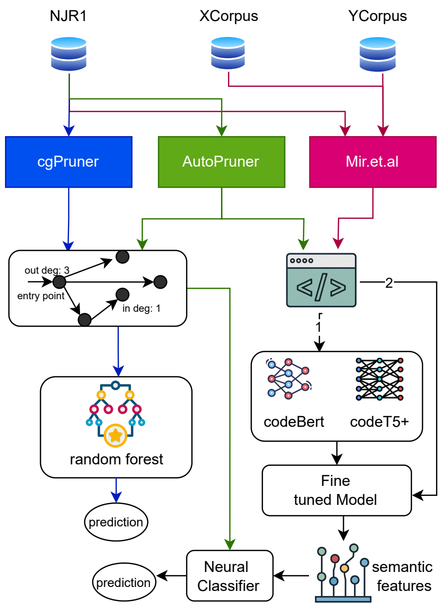


Figure 1: Overview of existing approaches.

Overall, this work makes the following contributions:

- We identify key limitations in existing evaluations of call graph pruning techniques, and empirically demonstrate that the reported results may be misleading.
- We create a new dataset suitable for training and evaluating the ML-based call graph pruning approaches, reusable for future research.
- We comprehensively setup an empirical re-evaluation of the call graph pruning approaches. The results show that while pruning models can slightly increase the precision of the call graphs, they significantly reduce the coverage.

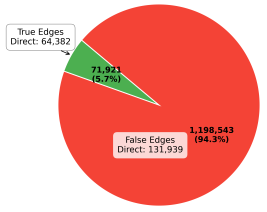
2 Are Existing Evaluations of CG Pruning Trustworthy?

In this section, we first provide the background of existing call graph pruning techniques and their evaluations. We then discuss and empirically demonstrate the limitations in these evaluations.

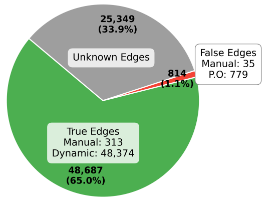
2.1 Existing CG Pruning Approaches

While there are various studies on applying machine learning to prune false alarms in static analysis (discussed in Section 7) or leveraging data- and control-flow patterns to improve the accuracy of call graph or pointer analysis [25, 32, 35, 54], to the best of our knowledge, there are three existing approaches that specifically apply machine learning to target call graph pruning [31, 38, 56].

Figure 1 shows an overview of the three existing approaches for call graph pruning, including the ML model architectures and the used datasets. **cgPruner** was the first approach in this direction, proposed by Utture et al. [56]. Its core idea, which is also adopted by subsequent work, is to train using the features of each call graph edge to predict the likelihood of the edge to be a true or false



(a) Original NJR1.



(b) Updated labels.

Figure 2: Original NJR1 dataset vs. updated labels.

positive, and then a threshold on the confidence of the prediction is used to prune the likely false edges from the call graph. The workflow of cgPruner is shown in the left of Figure 1, connected by the blue edges. For cgPruner, 11 structured features, extracted from the static call graph, were used as the input features of a random forest model [18], consisting of features local to an individual edge (e.g., number of edges ending in the caller node) and global to the entire call graph (e.g., total number of nodes in the call graph).

Another important contribution of cgPruner is the introduction of a new dataset, **NJR1**, for training and evaluating its approach. To generate this dataset, first, the call graphs of a subset of programs (141) in the NJR1 benchmark [44] were generated using the context-insensitive analysis of WALA [19]. Reflection handling was disabled, as the authors observed that enabling reflection handling in WALA's call graph construction introduced many false-positive edges and argued this presents no additional challenge for call graph pruning. All JVM library calls in generated call graphs were removed, because the ML models may overfit to the large amount of such edges if included. To preserve their effect on the overall program flow, the transitive impact of those calls was retained. Specifically, if there existed a call from an application method to a JVM method, then if there exists a path from that JVM method to another application method, a transitive edge was added between the two application methods in the call graph.

The dataset was constructed by labeling the edges in the static call graph using dynamic call graphs from the NJR1 programs, generated using a tool called *WireTap* [24], as ground truth. The edges not covered in the dynamic call graphs are considered false. Figure 2a shows the label distribution of this dataset. It contains a total of 1,270,464 edges, the majority (94.3%) of which are (assumed as) false edges. Among these, transitive edges dominate in both quantity and proportion of false edges.

Following cgPruner, Le-Cong et al. [31] presented **AutoPruner**, which introduced a new set of input features, called *semantic features*, for improving the performance over cgPruner. As shown in the right part of Figure 1, connected by black edges, these features are derived by fine-tuning the codeBERT model [12] on the source code of the caller and callee methods, using the edge labels of the NJR1 dataset as supervision. First, the concatenated source code of the caller and callee methods is tokenized to generate a sequence of tokens. This token sequence, along with its corresponding label, is input to the pre-trained codeBERT model for fine-tuning (the edge marked as 1). After fine-tuning, the tokenized source code of each edge is passed through the fine-tuned model to generate a vector representation, referred to as the *semantic features* of the edge (the edge marked as 2). Their approach extends cgPruner [56] by concatenating the semantic features with cgPruner's structured features (the green edges in Figure 1) and training a neural network classifier on the combined representation to predict edge labels. AutoPruner also used the same NJR1 dataset for training.

More recently, **Mir et al. [38]** conducted a study evaluating two additional large language models, CodeT5 and CodeT5+ [57], for pruning call graphs. They followed the same process of fine-tuning these model as AutoPruner did (thus, sharing the workflow connected by the black edges in Figure 1), but the structured features are not used in their models. They expanded the original cgPruner's NJR1 dataset by incorporating 76 programs from XCorpus [10] and YCorpus [27], and combined all three datasets into a benchmark called NYXCorpus. The programs in XCorpus and YCorpus are real-world Java programs. The NYXCorpus followed the same labeling process as the NJR1 dataset, using the dynamic call graph as the ground truth. The ratio of false to true edges are 2.4 and 3.7 for XCorpus and YCorpus, respectively, which is different from the NJR1 dataset. It was claimed that the NYXCorpus improves the generalization across diverse program types.

2.2 Limitations in Existing Evaluations

The datasets and the evaluations of the approaches discussed in Section 2.1 all rely on the assumption that the dynamic call graphs are adequate to be used as the ground truths. However, this assumption may not hold true, if the labeled datasets do no represent the actual distribution of the true and false call graph edges. While this threat to the validity is discussed in each work, the validity of the results was not explicitly assessed.

2.2.1 Labeling NJR1 dataset. To evaluate the impact of this assumption in the NJR1 dataset, which is used in all the existing approaches, we update this dataset to produce actual validated true and false edges. To prepare for the label update, we first filtered out the transitive edges from NJR1, as they do not directly reflect actual method invocations in the program. These edges contributed to the majority of the (assumed) false edges in the dataset. We also removed all of the library calls from the dataset in order to examine the call graph edges of application code. As a results, we are left with 74,850 edges in the updated dataset within which 48,374 are shown in the dynamic call graph and are labeled true. Next, we adopt the concept of partial orders between analysis options, as introduced by Mordahl et al. [41], to automatically label a subset of false edges. Specifically, we generated call graphs for the programs in the NJR1 dataset using WALA with the 1-call-site-sensitive analysis which is a more precise configuration—one that maintains the same level of soundness and forms a precision partial order with the original context-insensitive analysis used to construct NJR1. By comparing the two call graphs, we collected all edges that appear in the original call graph but are absent from the more precise one, and labeled these as false edges. We identified 779 false edges using this approach. The validity and detail of this labeling method using partial orders between analysis options is discussed in Section 3.2.2.

To label the remaining 25,697 unknown edges, we sampled 350 based on a 95% confidence level and 0.05 margin of error. One author labeled them, and another verified the labels independently. Labeling took 30 hours, with 6 additional hours for verification.

Table 1: Precision and coverage of original and pruned call graphs using AssumeNJR1 and ValidNJR1.

Variation	Original CG		Technique	Pruned CG	
	Prec	Cov		Prec	Cov
AssumeNJR1	0.54	0.71	cgPruner	0.67	0.61
			AutoPruner	0.64	0.63
ValidNJR1	0.97	0.71	cgPruner	0.96	0.71
			AutoPruner	0.98	0.71

Eventually, we were able to label 348 edges out of 350 sampled edges (2 undecided). Details of the labeling protocol are discussed in Section 3.2.3, where we present a new, more comprehensive dataset. Surprisingly, 313 out of 348 manually labeled edges are true. Note that all these edges are assumed to be false in the original NJR1 dataset, indicating a significant amount of incorrect labels in this dataset. Figure 2b shows the updated labels of NJR1, with 65% and 1.1% edges as true and false, and the rest (33.9%) unknown.

2.2.2 CG Pruning Using Updated NJR1 Labels. To demonstrate the impact of this mislabeling, we empirically compare the precision and coverage of the pruned call graphs (replicating cgPruner and AutoPruner) using the updated NJR1 datasets. Precision is the ratio of true edges in the pruned call graph to the total number of edges in the pruned call graph, and coverage is the ratio of true edges in the pruned call graph to the total number of true edges in the dataset. We introduce two variations in this empirical study, differed by the dataset labels used for training and testing. First, replicating how the existing evaluations [31, 38, 56] treat unknown labels, we call the dataset that assumes the 25,349 unknown edges in updated NJR1 as false, the *AssumeNJR1*, consisting of 48,687 true and 26,213 (25,349 + 814) false edges. Second, the dataset that only uses the validated labels (i.e., 48,687 true and 814 false edges) in the updated NJR1 is called the *ValidNJR1*.

We performed a 5-fold validation and report the average results in Table 1. Because of the mislabeling, using *AssumeNJR1* would result in an original call graph that has a low precision (54% in column 2), leaving the impression that this call graph has a lot of room for improvement. Applying cgPruner and AutoPruner, both techniques improve the precision by over 10% (column 5), though this improvement comes at the cost of a similar reduction in coverage (column 6).[1] Using *ValidNJR1* paints a completely different picture. The original call graph already has a high precision of 97%, with little room for improvement. In this setting, AutoPruner slightly improves the precision further (to 98%), while cgPruner actually decreases it (to 96%)—yet both models manage to preserve the overall coverage. Overall, our empirical findings highlight three important limitations in existing datasets and evaluations:

(1) Using dynamic call graphs as ground truths and training ML models on incorrect labels lead to unreliable results.
(2) The NJR1 programs are not sufficiently complex to cause static analysis tools to generate a substantial number of false edges, hereby limiting utilization of CG pruning.
(3) The analysis configurations used in prior work, particularly those with reflection handling disabled, fail to capture the full picture of analysis imprecision.

[1]Note that the performance improvement reported in the original evaluations [31, 38, 56] is even more significant than what we reported, due to including the transitive and library edges in NJR1.

Table 2: Overview of our dataset and its labels. P.O means Partial Orders.

Program	LoC	Dynamic	P.O.	Manual	
		True	False	True	False
Axion	24,113	5,142	1,992	135	90
Batik	179,129	18,407	6,656	129	96
Jasml	5,595	2,410	155	143	55
Xerces	192,110	5,438	1,004	78	22
Total		31,397	9,807	485	263

3 Dataset Creation

Motivated by the above limitations we identified, we aim to conduct a sound and comprehensive empirical re-evaluation to understand the state-of-the-art of call graph pruning techniques. The first step towards this study is the creation of a dataset that needs to satisfy: (**C1**) consisting of real-world programs representing the challenging nature of predicting calling relationships, and (**C2**) containing both validated true and false call graph edges, suitable for training, validating, and testing the ML-based models.

3.1 Target Programs

The XCorpus benchmark [10] consists of 76 executable, real-world Java programs. Due to its executable nature and active maintenance, it has gained traction in recent static analysis literature [15, 38]. Compared to benchmarks such as DaCapo [4], XCorpus offers more up-to-date programs with higher dynamic coverage [10], making it a preferred choice for empirical studies. Helm et al. [15] selected four programs from the XCorpus benchmark and refined their test harnesses to evaluate the soundness of call graphs produced by various static analysis tools. Their selection was guided by practical criteria: the chosen programs can be executed on a single machine without external dependencies, do not require GPU access or network connectivity, accept structured input formats, and span diverse application domains. *Axion* is a SQL database engine that processes SQL commands, *Batik* is a toolkit for SVG processing, *Jasml* is a Java assembler and compiler, and *Xerces* is a widely used XML parser. Column 2 of Table 2 shows the lines of code of each program. These real-world programs are good candidates for evaluating the effectiveness of call graph pruning, satisfying **C1**.

3.2 Labeling Process

To correctly create validated true and false edges for our dataset (**C2**), we applied three complementary approaches of labeling: (1) using dynamic call graphs, (2) applying analysis option partial orders, and (3) manual labeling. In this dataset, we focus on the call graph edges that are related to the applications themselves instead of JVM or the third party libraries. Therefore, we only consider the application edges, which have at least one of the methods (either the source or the target, or both) belonging to the target program and discard invocations between two library methods.

3.2.1 *Dynamic Call Graphs.*

For the four target programs, Helm et al. [15] manually inspected their source code to refine the harnesses (i.e., program entry points), and applied *Jazzer* [9], a greybox fuzzer for Java programs based on LibFuzzer [33], to achieve high code coverage. According to their findings, these improvements increased branch coverage by 2–13% across the selected programs. We adopt their results and utilize the improved dynamic call graphs as part of the validated true labels in our dataset. As shown in column 3 of Table 2, 31,397 edges are labeled true using this approach.

3.2.2 *Analysis Option Partial Order.*

While the dynamic call graphs allow automatically labeling a subset of true edges, we are not aware of any existing work that automatically labels false call graph edges in a dataset. To mitigate this challenge, we adapt the idea of analysis option partial order, introduced by Mordahl et al. [41], to label a subset of false edges. The concept of analysis option partial orders was originally introduced to identify bugs in static analysis tools. *Precision partial orders* compare a pair of configurations with equal soundness levels, expecting that the more precise configurations should produce results that are subsets of those from the less precise one. Deviations from this expected behavior may indicate flaws or inconsistencies in the analysis. In our work, we adapt the idea of precision partial orders to identify potential false positives in static call graphs. Specifically, we examine edges that appear in a less precise configuration but are absent from the more precise one. Such edges are treated as likely false positives. We discuss the limitation of this adaption in Section 6.

We selected three call graph analysis frameworks, WALA [19], Doop [5], and OPAL [16], for this purpose. Each tool provides a range of configuration options and handles various language features differently. For example, WALA offers multiple settings for handling reflection, while Doop provides a binary option (enabled or disabled) for the same feature. For each of the tools, we identified a set of options to construct a configuration space. For WALA and Doop, we utilized the set of configuration options proposed by Mordahl et al. [40] (4 options and 27 settings for WALA and 19 options and 66 settings for Doop) that directly impacted the precision and soundness, while for OPAL, we analyzed the tool's source code and available documentation (1 option and 10 settings).

Exhaustively running all possible configurations was not feasible with limited resources. Therefore, we chose configurations that are most relevant for our experiments. Our configuration selection criteria prioritized configurations that (1) produce partial orders with other configurations, and (2) offer diversity in terms of soundness and precision levels. The process began with the default configuration of each tool, after which we modified one option at a time that contributed either to the soundness (e.g., reflection handling) or precision (e.g., context sensitivity) of the analysis. In some cases, we adjusted two options simultaneously in order to ensure greater diversity among the selected configurations. For example, for WALA, we changed the reflection settings as well as handling static initializers which both contribute to the soundness level of the analysis. As a result, we identified 88, 81, and 10 configurations for WALA, Doop, and OPAL, respectively. We executed each configuration on the four target programs, applying a timeout of three hours. We included these configurations and the generated call graphs in our artifact [46].

Using the generated call graphs, we apply the precision partial order relationship to automatically label the false call graph edges. The set of configurations that we chose that directly contributed to the precision of the analysis are context-insensitive (WALA, Doop), 1/2-call-site-sensitive (WALA, Doop), 1/2-object-sensitive (WALA, Doop), 1/2-type-sensitive (Doop), and 0/1-heap-sensitive (Doop) analyses. Each tool have different variations of these configurations. Specifically, we compared between a pair of configurations only if k differs for the same algorithm. For Example, 2-object-sensitive was compared to 1-object-sensitive, but not to 1-call-site-sensitive. For each pair of configurations we compared, the other options always have the same settings (e.g., same reflection setting), ensuring the same level of soundness. We did not include settings that Mordahl et al. [41] and Helm et al. [15] reported different soundness (CHA, RTA) or violations. Because OPAL's options are limited, we did not use it for this labeling process. By comparing the two call graphs generated by each pair of configurations, we collected all edges that appear in the less precise call graph but are absent from the more precise one, and labeled these as false edges. In total, 9807 false edges were generated using this approach, shown in column 4 of Table 2.

3.2.3 *Manual Labeling.*

While useful, the above approaches cannot automatically label all call graph edges. Therefore, we seek manual labeling to produce additional validated labels. To achieve this, we first need to decide in which call graphs we label additional edges. Among the call graphs we generated in Section 3.2.2, we computed the number of true (labeled by the dynamic call graphs), false (labeled by the analysis option partial orders), and unknown edges for each call graph. Based on this information, we selected call graphs generated by configurations that are diverse in terms of the number of edges they contain. Eventually, we selected three configurations each for WALA and Doop, and one configuration for OPAL. Due to OPAL's limited support for reflective calls, its generated call graphs were often completely devoid of application edges. Therefore, we included only the configuration that set the *CHA* option with reflection handling disabled (**OC1**). The Doop configurations are (1) context-insensitive analysis with reflection handling enabled (**DC1**), (2) context-insensitive analysis with reflection handling disabled (**DC2**), and (3) 1-type-sensitive analysis with reflection handling enabled (**DC3**). The WALA configurations are (1) context-insensitive analysis with *ONE_FLOW_TO_CASTS_NO_METHOD_INVOKE* as the reflection level (**WC1**), (2) 1-call-site-sensitive analysis with *ONE_FLOW_TO_CASTS_NO_METHOD_INVOKE* (**WC2**), and (3) context-insensitive with *String_only* reflection handling (**WC3**).

Because it is not feasible to label all the unknown edges in these call graphs, we select a subset of representative edges to label. We chose the two-level stratified sampling strategy [50] to randomly sample the subset. To prepare for sampling, we first divide the entire dataset by program, allocating a fixed portion of the total sample budget to each one. This guarantees every program is equally represented. For the data within each program, we further stratify the edges into two sub-groups: (1) the *Intersection set*, consisting of unknown edges that appear in the call graphs of all configurations for a given program, and (2) the *Symmetric difference set*, comprising unknown edges that appear in at least one configuration but not all. These sets capture two complementary views: the Intersection set ensures coverage across all configurations, while the Symmetric difference set captures configuration-specific edges. We chose the two-level stratification strategy over simple random sampling because random sampling from the global pool of unknown edges could result in an unbalanced sample, where some call graphs may receive too few labeled edges to support meaningful evaluation.

To label each edge, we follow two steps to decide its label. First, we check the reachability of the source method (i.e., caller) of the call graph edge from the entry point of the program. To verify this, we check if the source method is in the dynamic call graph. If so, the call is reachable. Otherwise, we use IDEs to find the methods that call the source method. If those methods are in the dynamic call graph and it is possible that the source method is called then it is reachable. We do this iteratively until timeout (10 minutes) or until we can verify that it is reachable or not. Regex matching is used for reflective calls explicitly using the method names. Second, we check the logic of the call site. For an invocation within conditional branches, we label it true only if it is certain that there exists an execution path where the branch condition holds and the invocation can occur; otherwise, it is false. For dynamic dispatch, the label is determined by the compatibility of the receiver object's type with the target method's declaring class. To do this, labelers trace from the receiver object following backward dataflow through assignments and parameter passing across methods, to possible allocation sites. We classify the edge as false if the allocation trace of callee's type cannot be found. Lastly, if the signature of the callee does not match the actual method that should be invoked, the edge is labeled false. If all the logic checks pass, we label the edge as true.

Three labelers manually labeled 748 edges. A static analysis expert labeled 200, while two trained Java-proficient labelers each independently labeled the remaining 548, resolving disagreements with the expert's help. The Cohen's kappa agreement rate was 0.66%, indicating *substantial* agreement [30]. In total, we labeled 485 true and 263 false edges shown in columns 5 and 6 of Table 2. Note that we intentionally sampled and labeled fewer edges for *Xerces*, because only three out of the seven configurations completed within the timeout on this program. Overall, this process took approximately 80 person-hours in three weeks, including training and conflict resolution. In Table 2, we observe that the ratios of true to false labels differ between automatically (3:1 in columns 3 and 4) and manually (2:1 in columns 5 and 6) labeled edges.

4 Study Setup

Using the created dataset, our empirical evaluation aims to answer the following research questions:

- **RQ1:** How effective are the state-of-the-art machine learning-based call graph pruning approaches?
- **RQ2:** Can the pruning approaches be generalized across tools and configurations?
- **RQ3:** What is the impact of model variations, including data balancing and splitting?

4.1 Studied Call Graphs and Metrics

To evaluate the state-of-the-art approaches in pruning call graphs, we aim to study diverse call graphs. These call graphs should vary in their precision and coverage to allow understanding the applicability of the pruning approaches in different scenarios. Specifically, we focus on call graphs generated by three popular call graph analysis frameworks: WALA, Doop, and OPAL. We reuse the 7 configurations (3 for WALA, 3 for Doop, and 1 for OPAL) that were used for manually labeling our dataset to ensure that validated labels are available when evaluating the pruning approaches. The precision and coverage of these call graphs are shown in columns 2-5 of Table 3. Note that we compute these metrics separating the labels by those generated automatically and those generated by manual labeling. This is because (1) we observe different label distributions between different approaches as discussed in Section 3.2.3, and (2) we use the automatically generated labels for training and validation and the manual labels as a holdout dataset for testing.

We compute the precision and coverage of the call graphs as follows: Precision = $\frac{|S_{db} \cap T_{db}|}{|S_{db}|}$ and Coverage = $\frac{|S_{db} \cap T_{db}|}{|T_{db}|}$, where S_{db} denotes the set of all edges in the call graph, and T_{db} represents the corresponding set of true edges, depending on the evaluation set. The variable *db* can be either *auto* (for the automatically labeled set) or *manual* (for the manually labeled set). After computing precision and coverage for each program under a given configuration, we report the average values across all programs for each configuration to capture the overall results of the call graph for that configuration.

In Table 3, the Auto results indicate that precision ranges from 0.82 to 0.89 (column 2), while in the Manual set, precision varies more significantly—from 0.49 to 0.78 (column 4). This disparity highlights the differing characteristics between the auto and manual sets, as well as the variability in call graph quality across configurations. Regarding coverage, the Auto results (column 3) range from 0.72 (**DC2** of Doop) to 0.91 (**OC1** of OPAL). For the Manual set, the coverage spans a similar range—from 0.79 to 0.96 (column 5).

4.2 Machine Learning Models

4.2.1 Compared Models and Feature Sets. We compare the models introduced in the literature (Section 2.1) and their variations.

Random Forest (RF). The random forest model was proposed by Utture et al. [56]. We employ the same hyperparameters specified in their work. This model was trained using 11 structured features, as discussed in Section 2.1.

codeBERT (Bert) and codeT5+ (T5). We study the two pre-trained models, codeBERT [12] by Le-Cong et al. [31] and codeT5+ by Mir et al. [38], employing the same hyperparameters. The process of extracting semantic features is described in Section 2.1.

Bert-NN and T5-NN. We observe that the fine-tuning process for Bert and T5 is computationally expensive, we therefore also utilize lightweight pre-trained Bert and T5 models without fine-tuning by tokenizing the call graph edges using the same procedure as in the fine-tuned models, except that we omit the label-guided fine-tuning step. Instead, we directly pass the tokenized edge representations to the pre-trained models. The resulting fixed-length encoded outputs for each edge are then used as input features to train a downstream neural network classifier for edge prediction.

All the above models produce a confidence score for each of their predictions, indicating how certain the model is about its output. An edge is predicted as *true* if the confidence score exceeds a predefined threshold; otherwise, it is predicted as *false*. We adopt a default threshold of 0.5 for all models, except for the random forest model, for which the threshold is set to 0.45, replicating the settings used in the existing evaluations [31, 38, 56].

4.2.2 Model Training. To answer the three RQs, we train different variations of the 5 models discussed above.

RQ1. To evaluate the effectiveness of the call graph pruning techniques, consistent with the existing work's setup [31, 38, 56], in this RQ, we train one model for each tool's configuration. For this and the next RQ, we adopted a program-wise splitting strategy, in which entire programs were used only for training or only for testing. In this setup, we trained each model using a leave-one-out approach: training on two or three programs (depending on the total number available call graphs for that program) and testing on the remaining program. We applied 3-fold or 4-fold cross-validation to ensure that each program in the dataset served as the test set at least once. For example, for the 4 call graphs generated by the Doop configuration *DC1*, each time we train on 3 call graphs using one of the 5 models and evaluate this model on the remaining call graph. In total, with 7 configurations, 5 model architectures, and cross-validation, we constructed 120 models for comparison to answer RQ1. We balanced the data using the oversampling technique and ratio of 1. We use this data balancing setting because it performs the best among all settings we experimented with (discussed below).

RQ2. To evaluate the generalizability of the call graph pruning techniques when applied across tool configurations or even tools, in this RQ, we set up two additional variations on how the machine learning models are trained. First, we train a model using all call graphs generated by a specific tool (e.g., Doop) across all training programs. For testing, we apply the trained model to prune the call graphs of a previously unseen program. This setup allows us to assess whether a model trained on the diverse characteristics of all configurations of a tool can generalize well enough to prune the call graph of a new program as effectively as a model trained specifically on one configuration. Since only one configuration was selected for OPAL, it was excluded from this experiment. This setup yielded 15 and 20 models corresponding to WALA and Doop, respectively.

Second, we train a general-purpose model, training on call graphs generated by all configurations of all tools (*WALA*, *Doop*, and *OPAL*) using only the training programs. The evaluation is on call graphs from the testing program, regardless of the tool or configuration used to generate them. This setup allows us to evaluate the model's ability to generalize beyond a specific tool or configuration.

RQ3. RQ3 explores the effect of different data splitting and balancing settings. Regarding data splitting, we compare two scenarios. First is the program-wise splitting (same as in RQ1 and RQ2), simulating a real-world scenario in which no labeled edges are available for the program being pruned, and the model must generalize to previously unseen programs. Second, we performed random edge-level splitting, where edges were randomly divided into training and validation sets. The models are created using 5-fold cross-validation. Note that this data splitting approach would have data points from the same call graph in both the training and validation dataset, mimicking a scenario in which the call graph pruning is applied to a call graph with a subset of edges labeled and used for training.

Regarding data balancing, as shown in Table 2, the labels in our dataset are unbalanced, with many more true labels than the false ones. To assess its effect, we employ two solutions: downsampling and oversampling [39]. Downsampling reduces the number of true edges by randomly selecting them for reduction, while oversampling increases the number of false edges by duplicating existing false edges. For each technique, we experimented with the balancing ratios 0.1, 0.25, 0.5, 0.75, and 1.0. Due to the expensive nature of fine-tuning Bert and T5 models, as we discussed above, we only applied these balancing techniques to the RF, Bert-NN, and T5-NN.

Experimental Environment. To generate the dataset, we used a server equipped with an Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz and 377 GB of RAM. For training the machine learning models, we used Google Colab Pro with an NVIDIA A100 GPU (40 GB VRAM) and 83 GB of RAM.

5 Study Results

In this section, we present the results of our empirical study, structured around the three research questions outlined in Section 4.

5.1 RQ1: Effectiveness of CG Pruning

Table 3 presents the call graph results of RQ1. Columns 7-10 show the precision and coverage of the pruned call graphs, separated by the automatically labeled (validation) and manually labeled (holdout) datasets. For both the validation and holdout results, we observe that *the pruning models are not effective. While they often can increase the precision of the call graphs, this comes at the cost of significantly reduced coverage across all models*, which could lead to diminished utility in downstream tasks [15, 47, 49].

Among all the models, Bert achieved the highest precision gain of 9% on the validation set for configuration *WC1*, at the cost of a 12% reduction in coverage. Overall, *random forest and fine-tuned models (Bert and T5) perform better in preserving coverage compared to the Bert_NN and T5_NN.* In particular, *RF* often achieves the highest coverage, retaining many true edges. Fine-tuned models, on the other hand, tend to prune more aggressively, typically producing call graphs with higher precision at the expense of coverage. Conversely, *Bert_NN* and *T5_NN* exhibit significant reductions in coverage across all configurations, indicating a tendency to over-prune and exclude a substantial portion of true edges. For configuration *WC2*, using *Bert_NN*, precision of the pruned call graph is lower than that of the original one. This is because pruning reduces the total number of edges, and if the model mistakenly removes many true edges, the resulting precision can drop below the original.

Next, we evaluate the performance of the models, shown in Table 4. Based on the *validation* results (columns 3-5), the fine-tuned models achieved the highest recall in most configurations, while *T5_NN* consistently yielded the highest precision. The F1 score, however, varies across different models and configurations, with *Bert_NN* and *T5_NN* consistently showing lower recall than other models across different configurations.

The holdout results of the models differ significantly from validation results. We observe that precision of the models is generally lower on the holdout set compared to the validation set, while recall remains relatively stable. For instance, for configuration *OC1*, *RF* only has a precision of 60% on the holdout set, compared to 82% on the validation set. This discrepancy suggests that *manually labeled edges exhibit substantially different characteristics compared*

Table 3: Precision and coverage of original and pruned call graphs for each configuration. Configurations starting with D for Doop, W for WALA, O for OPAL. Auto means automatically labeled set and Manual means manually labeled set. Pre columns show Precision results, and Cov columns show Coverage results.

CID	Original CG				Model	Pruned CG			
	Auto		Manual			Auto		Manual	
	Pre	Cov	Pre	Cov		Pre	Cov	Pre	Cov
DC1	0.82	0.88	0.66	0.93	RF	0.88	0.83	0.66	0.78
					Bert_NN	0.85	0.59	0.67	0.68
					T5_NN	0.89	0.54	0.63	0.53
					Bert	0.89	0.78	0.67	0.80
					T5	0.86	0.75	0.67	0.79
DC2	0.88	0.72	0.78	0.79	RF	0.89	0.70	0.79	0.76
					Bert_NN	0.89	0.38	0.84	0.44
					T5_NN	0.92	0.44	0.76	0.50
					Bert	0.92	0.67	0.79	0.73
					T5	0.93	0.65	0.80	0.75
DC3	0.89	0.88	0.66	0.93	RF	0.89	0.85	0.64	0.81
					Bert_NN	0.92	0.54	0.66	0.55
					T5_NN	0.93	0.63	0.66	0.62
					Bert	0.91	0.83	0.67	0.90
					T5	0.91	0.84	0.66	0.89
WC1	0.80	0.84	0.75	0.92	RF	0.85	0.77	0.75	0.82
					Bert_NN	0.83	0.59	0.76	0.73
					T5_NN	0.87	0.45	0.78	0.44
					Bert	0.89	0.72	0.80	0.81
					T5	0.87	0.73	0.78	0.83
WC2	0.88	0.84	0.75	0.92	RF	0.91	0.82	0.75	0.86
					Bert_NN	0.87	0.58	0.77	0.69
					T5_NN	0.92	0.51	0.79	0.61
					Bert	0.94	0.76	0.79	0.82
					T5	0.92	0.74	0.78	0.85
WC3	0.82	0.84	0.75	0.92	RF	0.86	0.82	0.76	0.87
					Bert_NN	0.87	0.55	0.78	0.60
					T5_NN	0.88	0.50	0.79	0.58
					Bert	0.88	0.79	0.77	0.90
					T5	0.88	0.76	0.78	0.84
OC1	0.85	0.91	0.49	0.96	RF	0.87	0.87	0.56	0.88
					Bert_NN	0.86	0.66	0.50	0.69
					T5_NN	0.89	0.59	0.53	0.56
					Bert	0.89	0.86	0.55	0.88
					T5	0.87	0.87	0.53	0.88

to automatically labeled ones. Since the models are trained on automatically labeled data, they fail to correctly classify edges from the unseen set in the dataset. This observation also implies that *the features proposed in prior work may not be sufficient to effectively distinguish between all true and false edges.*

When comparing model performance metrics (Table 4) with the resulting pruned call graphs (Table 3), we find that higher model precision and recall do not always translate to better precision and coverage in the call graphs. For instance, for configuration *DC2*, *Bert_NN* achieved a higher recall (98%) than *RF* (94%) in validation results. However, the coverage of the pruned call graphs tells a different story: *Bert_NN* achieved only 67% coverage, while *RF* achieved 70%. The reason for this discrepancy lies in the difference between how model recall and call graph coverage are calculated. As discussed in Section 4, coverage is computed based on the set of true edges in each program, whereas model recall does not require knowledge of the full set of true edges in the program. Instead, it is based on the model's predictions relative to a labeled test set. Previous work [31, 38, 56] has primarily focused on pruned call graphs' results and has not reported model performance metrics.

Table 4: Model performance. Configurations starting with D for Doop, W for WALA, O for OPAL. Auto means automatically labeled set and Manual means manually labeled set. Pre columns show Precision results, and Rec columns show Recall results.

CID	Model	Auto (Validation)			Manual (Holdout)		
		Pre	Rec	F1	Pre	Rec	F1
DC1	RF	0.84	0.91	0.87	0.75	0.90	0.82
	Bert_NN	0.79	0.62	0.70	0.72	0.73	0.73
	T5_NN	0.83	0.58	0.68	0.73	0.59	0.65
	Bert	0.82	0.86	0.84	0.75	0.87	0.81
	T5	0.79	0.92	0.85	0.72	0.85	0.78
DC2	RF	0.83	0.94	0.88	0.81	0.97	0.89
	Bert_NN	0.82	0.53	0.64	0.83	0.54	0.65
	T5_NN	0.86	0.57	0.68	0.82	0.66	0.74
	Bert	0.84	0.98	0.90	0.80	0.93	0.86
	T5	0.86	0.88	0.87	0.82	0.96	0.88
DC3	RF	0.84	0.96	0.90	0.73	0.94	0.82
	Bert_NN	0.88	0.52	0.65	0.73	0.55	0.63
	T5_NN	0.89	0.66	0.76	0.74	0.65	0.69
	Bert	0.86	0.98	0.91	0.72	0.96	0.82
	T5	0.86	0.97	0.91	0.72	0.97	0.83
WC1	RF	0.80	0.93	0.86	0.76	0.90	0.82
	Bert_NN	0.77	0.68	0.72	0.78	0.79	0.78
	T5_NN	0.80	0.55	0.65	0.78	0.47	0.59
	Bert	0.79	0.93	0.86	0.79	0.87	0.83
	T5	0.79	0.93	0.85	0.77	0.90	0.83
WC2	RF	0.88	0.94	0.91	0.76	0.95	0.84
	Bert_NN	0.84	0.71	0.77	0.76	0.74	0.75
	T5_NN	0.88	0.64	0.74	0.82	0.67	0.74
	Bert	0.88	0.94	0.91	0.79	0.88	0.83
	T5	0.87	0.95	0.91	0.77	0.93	0.84
WC3	RF	0.80	0.95	0.87	0.77	0.95	0.85
	Bert_NN	0.78	0.63	0.70	0.78	0.64	0.70
	T5_NN	0.81	0.57	0.67	0.82	0.64	0.72
	Bert	0.80	0.97	0.87	0.78	0.99	0.87
	T5	0.79	0.95	0.86	0.78	0.92	0.84
OC1	RF	0.82	0.95	0.88	0.60	0.94	0.73
	Bert_NN	0.80	0.64	0.71	0.55	0.72	0.62
	T5_NN	0.83	0.63	0.71	0.57	0.61	0.59
	Bert	0.82	0.95	0.88	0.59	0.91	0.71
	T5	0.80	0.96	0.88	0.57	0.93	0.70

To better understand the behavior of the models, we conducted an in-depth analysis on the incorrect predictions made by the RF and Bert models on the validation set for configurations *DC1* and *OC1*. *RF* achieved a precision of 82% and recall of 95% for *DC1*, while *Bert* achieved a precision of 82% and recall of 96%. The results of this analysis reveal that these models produced significantly different

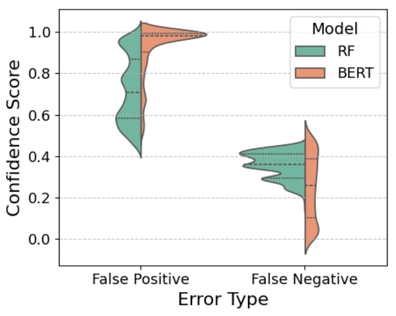


Figure 3: Confidence distribution of the RF and Bert models on the validation set for misclassified edges.

false negatives, while they produced many shared false positives. For configuration *OC1*, only 10% of the false negatives from *RF* and 11% from *Bert* are shared. In contrast, 68% of the false positives for *RF* and 70% for *Bert* are shared between the two models. This suggests that the two models are likely capturing complementary patterns in what they prune, but tend to agree on the predictions they mistakenly identify as true edge. This result indicates that false positives are systematically harder to distinguish based on the available features, or that both models are sensitive to similar misleading patterns.

Through manual inspection of these false positives and false negatives, we found that the *Bert* tends to struggle with edges involving constructors and class initializers, while *RF* is not accurate on predicting dynamic calls that result in multiple outgoing edges from a single call site. Class initializers are often generated by the compiler and may not be explicitly defined in the source code, making them challenging for the *Bert* model to learn the semantic features. As for *RF*, it uses structured features such as *L-fanout*, which measures the number of outgoing edges from a call site. Such features are unable to indicate which outgoing edges are correct, as they do not provide fine-grained distinctions between true and false edges when a call site has multiple targets.

We also analyzed the confidence of the models' predictions for the misclassified edges, presented in Figure 3. The results show that *Bert* tends to be highly confident about its false positives, whereas the distribution of confidence scores for *RF*'s false positives is more uniform. In contrast, for false negatives, Bert exhibits a more uniform confidence distribution, while RF generally shows low confidence in its incorrect predictions. This indicates that *Bert* may be overconfident in certain misleading patterns, potentially due to its strong memorization or representation biases. Meanwhile, *RF* appears more uncertain when it misclassifies, which could make it more amenable to threshold-based filtering.

5.2 RQ2: Generalizing CG Pruning Models

Figure 4 presents the results of this research question. The y-axis shows the average precision and coverage of pruned call graphs across the auto/validation and manual/holdout sets, for models trained on all configurations within each tool (tool in orange), as well as a general model trained across all tools (general in blue). For comparison, we also include models trained on single configurations (config in green), as in RQ1. The x-axis lists the pairs of configuration and model.

Interestingly, in most cases, *models trained on multiple configurations outperform those trained on a single configuration*, indicating that generalization is possible. Notably, the *general* model achieved the highest scores across all configurations of WALA and OPAL, with the exception of *RF* in the OC1 configuration. For Doop's configurations, we observe that the more general models either outperform or perform comparably to the configuration-specific ones. The most significant improvement is with *T5* in the DC1 configuration. Additionally, *Bert* and *T5* consistently outperformed *RF* under more generalized training settings. These findings suggest that general-purpose pruning models using Bert and T5 not only broaden the applicability of pruning across tools and configurations but also enhance the quality of the resulting call graphs.

Table 5: Model F1 with program-wise and random splitting.

Model	Auto (Validation)		Manual (Holdout)	
	Prog F1	Rand F1	Prog F1	Rand F1
RF	0.88	0.95	0.83	0.83
Bert_NN	0.79	0.82	0.76	0.74
T5_NN	0.79	0.83	0.74	0.76

5.3 RQ3: Impact of Data Balancing and Splitting

Figure 5 illustrates the performance trends across different balancing techniques and sampling ratios, ranging between 0.1 and 1. We observe that the performance of *RF* remains consistent across all balancing settings, suggesting that this model is robust to class imbalance. In contrast, *Bert_NN* and *T5_NN* are significantly affected by the class distribution and benefit from balanced datasets. Among the balancing strategies, oversampling consistently outperforms downsampling for *Bert_NN* and *T5_NN*. *Bert_NN* achieved its highest performance when trained on a dataset oversampled to a 0.25 false-to-true ratio. In comparison, *T5_NN* performed the best when false edges are oversampled to match the number of true edges, i.e., a ratio of 1. These results show that *data balancing impacts model performance, but the extent of this effect varies by model type. Overall, oversampling with a ratio of 1 yielded strong performance across all models, making it a generally effective strategy.*

The average F1 results of the models trained with program-wise split and random split are shown in Table 5. According to the *validation* results (columns 2 and 3), randomly selecting edges for training improves model performance across *Bert_NN* and *T5_NN*. *RF* shows the largest improvement, with an increase of 7% in F1 score, while the *Bert_NN* shows the smallest improvement at 3%. As for the *Holdout* set (columns 4 and 5), the results indicate little effect of the splitting strategy.

These results suggest that *the models heavily rely on program-specific call graph information.* When the training set includes edges from each target program, model performance improves, indicating that exposure to program-specific structures is beneficial. Conversely, when models are trained without any edges from certain programs (as in program-wise splitting), their ability to generalize

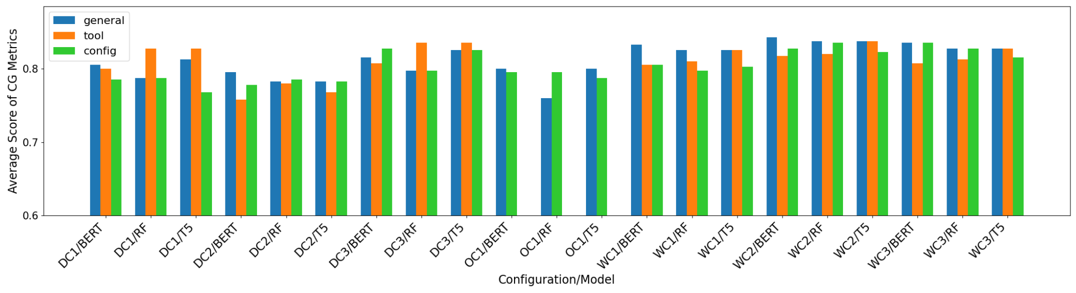


Figure 4: Results of pruning models trained across multiple tools and configurations. Configurations starting with D for Doop, W for WALA, O for OPAL. Tool (orange) indicates models trained on all configurations within a tool. General (blue) indicates models trained on all configurations across all tools. Config (green) indicates models trained on a single configuration.

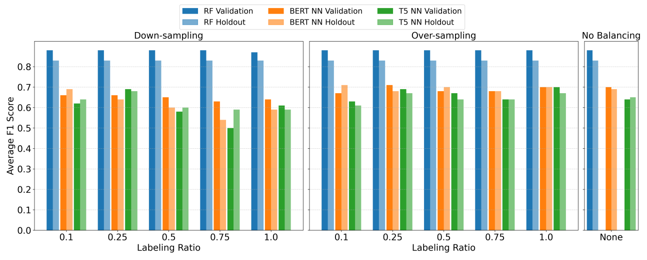


Figure 5: Data balancing results.

is limited. This implies that *the call graph pruning techniques are not sufficiently generalized to accurately prune edges from unseen programs*. Two possible explanations emerge: (1) call graph edges from different programs may exhibit unique, program-specific characteristics, suggesting that the feature representations proposed in previous work do not fully capture generalizable patterns across programs; and/or (2) the training data lacks sufficient diversity, leading to poor model performance on previously unseen edge types. In either case, the lack of representative features and labeled examples appears to be a key limitation affecting generalization.

6 Implication and Threats to Validity

Implication: open challenges and future directions. Our empirical re-evaluation highlights open challenges for the task of call graph pruning. One challenge is the lack of representative, ground-truth datasets to train and evaluate the ML models. Despite the complementary approaches we applied to label call graph edges, the manual effort is significant and the data still exhibit different features across the automatic and manual labels, indicating under-representation of all features that may exist in call graph edges. We identify two promising directions to address this challenge. First, recognizing the difficulty of obtaining representative data, especially for validated false edges, we believe it is important to develop methods that work effectively with partially labeled data. We preliminarily explored a promising idea, using Positive-Unlabeled (PU) learning [62], which enables training with only a subset of positive (true) labels. Adapting the approach presented by Yang et al. [59], we applied HDBSCAN [36] to cluster edges and used a portion of the true edges to assign labels to each cluster. The results show that PU learning may yield high-precision pruned call graphs without relying on any validated false labels. For example, for the *DC2* configuration, using structured features, this approach achieved a precision of 92% and a coverage of 63% of the pruned call graph—comparable to results in Table 3. We included all the results of PU learning in our artifact [46]. Second, the literature should prioritize and continue generating datasets that contain more ground truths. For this challenge, we believe the direction Helm et al. [15] has taken is promising, utilizing fuzzing to improve the representativeness of the labeled dataset. While this research improved the overall quality of the dynamic call graphs, further enhancements—such as better harness generation [8, 34, 55] for fuzzers—could increase coverage and labeling accuracy even more.

Another challenge is that our findings indicate that the feature representations used in prior work are often insufficient for reliably distinguishing true call graph edges from false ones. Manual inspection of misclassified edges revealed that semantic features frequently lack necessary program context. In many cases, false positives arise because these features fail to capture the necessary inter-procedural information. For example, the actual type of a receiver object in a dynamic call is often determined in a different method than the one containing the call site. As a result, semantic features limited to only the source and target methods miss critical contextual cues needed for accurate classification. These limitations suggest that future research should focus on designing more context-aware and discriminative feature representations—especially those capable of capturing inter-procedural control- and data-flow program information.

Threats to validity. There are several threats to the validity of our study. One threat arises from our use of partial orders among static analysis configurations to identify false edges. As reported in by Modahal et al. [41], these partial orders are not always reliable due to bugs in the implementations of static analysis tools. Consequently, some edges labeled as false may, in fact, be true. However, we expect the number of such misclassified edges to be small; we randomly checked 50 samples of these edges and determined they were all correctly labeled as false. Another threat concerns the generalizability of our results given the limited number of programs evaluated. While our dataset includes only four programs, we selected them from diverse application domains to improve representativeness and mitigate this limitation. A third threat involves our manual sampling and labeling process. The sampled edges may not perfectly reflect the true distribution of edges in the call graphs. To address this, we employed a stratified sampling strategy that ensures representation from each call graph and configuration. Manual labeling is inherently prone to human error and bias, which could affect the reliability of our labeled data. To mitigate this, we trained annotators, provided detailed labeling guidelines, and conducted multiple rounds of review, including expert validation, to enhance label accuracy.

7 Related Work

Call graph evaluation. A number of studies investigated the soundness and precision of static call graphs [11, 15, 47–49, 53]. Reif et al. [47, 48] introduced a framework for benchmarking static call graph construction. They developed an synthetic benchmark suite consisting of small programs with known sources of soundness and precision issues—cases that are typically missed by existing static analyses. These benchmarks come with a known ground truth, enabling precise evaluation of static call graph quality. Helm et al. [15] extended this line of work by applying fuzzing to real-world programs to approximate ground-truth dynamic call graphs, which we adopted as part of the validated labels in our dataset. By comparing these with static call graphs, they assessed the soundness of several static tools across four real-world programs. Samhi et al. [49] applied a similar methodology to evaluate static call graph soundness in Android applications, emphasizing the challenges of reflection and dynamic behavior. While we leverage the comparison between static and dynamic call graphs to generate a portion of our labeled data, we do not treat the dynamic call graph as ground truth, due to its inherent incompleteness.

Machine learning in static analysis. Several studies have applied machine learning to reduce false alarms produced by static analysis tools [2, 14, 26, 28, 29, 60, 61]. Koc et al. [28] introduced an approach that uses backward slicing to extract code snippets related to static analysis warnings and trains a classifier to predict whether each warning is a true or false positive. This work was extended by Koc et al. [29] and Yerramreddy et al. [61], exploring more advanced learning techniques such as recurrent and graph neural networks (RNNs and GNNs), along with different program representations including abstract syntax trees (ASTs) and control flow graphs (CFGs). These studies relied on two datasets for training and evaluation of thier models. The first is the OWASP Benchmark [43], which contains a large number of reports from static analysis tools along with ground-truth labels. In addition, they manually labeled the warnings produced by FindSecBugs [13] on 14 real-world Java projects, yielding a total of 400 annotated reports. Kharkar et al. [26] proposed an approach using pre-trained language models, such as CodeBERT, to improve warning classification. Using a manually generated ground-truth dataset of programs and warnings for training, they showed that neural features with LLMs outperform structured features, yielding higher precision improvement (+15.13%). Similarly, Yang et al. [60] employed large language models (LLMs) to prioritize warnings for manual inspection. In this study, the authors used three datasets that had been manually validated either by themselves or by prior work. For example, one dataset contained roughly 6,000 true positives and 70,000 false positives. For many of these warnings, additional manual validation was unnecessary because they were confirmed as fixed in later versions of the respective projects. The key difference between our evaluation methodology and prior work lies in the datasets. Static alarm datasets can feasibly be labeled manually, since the warnings are sparse. In contrast, a program's static call graph contains many edges, making only using manual labeling infeasible.

Other studies have employed machine learning to enhance the precision and efficiency of call graphs and points-to analyses. [6, 7, 17, 20–23, 42, 58]. Oh et al. [42] introduced a Bayesian optimization framework for learning where to apply precision-enhancing techniques in static analysis. Heo et al. [17] proposed selectively introducing unsoundness in parts of the program deemed harmless, using anomaly detection and one-class SVMs. Jeong et al. [23] and Jeon et al. [20] introduced data-driven techniques for optimizing context sensitivity in pointer analysis. By representing programs in disjunctive form, their models learn heuristics that assign context sensitivity depth to individual methods, improving precision and scalability without manual tuning. These approaches aim to improve static analysis precision by selectively introducing unsoundness or context sensitivity, and evaluate the effectiveness using metrics such as runtime, call graph size (e.g., number of edges), and impact on downstream tasks, whereas our work focuses on improving the precision by pruning call graphs and evaluating the pruned call graphs using precision and coverage metrics. Bhuiyan [3] proposed Graphia, a GNN-based approach that improves recall in JavaScript call graphs instead of precision, by reducing false negatives using AST features.

8 Conclusions

Our study revisits the foundations of learning-based call graph pruning and reveals critical limitations in how these techniques have been evaluated. We show that prior work relies on dynamic call graphs as ground truth, despite their incompleteness, leading to incorrect labels and misleading results. To assess the state of the art, we create a new, more reliable dataset constructed from real-world programs using dynamic fuzzing, analysis partial orders, and manual labeling. Using this dataset, we retrain existing pruning models and systematically evaluate them across seven configurations of three widely used static analysis tools. We further investigate the effect of model generalization, and data splitting and balancing techniques. Our findings demonstrate that while pruning can slightly improve precision, it often significantly reduces coverage of call graph results. Moreover, models trained across multiple tools and configurations generalize better than configuration-specific ones. However, the models still struggle with unseen programs, indicating that the features used in prior work do not adequately capture the complexities of real-world code. This highlights the need for more sophisticated feature engineering and sound datasets.

9 Data Availability

We made our dataset, evaluation scripts and results available [46].

Acknowledgments

This work was supported in part by the National Science Foundation under grants CCF-2008905 and CCF-2047682.

References

[1] Lars Ole Andersen. 1994. Program analysis and specialization for the C programming language. (1994).

[2] George David Apostolidis, Ilias Kalouptsoglou, Miltiadis Siavvas, Dionysios Kehagias, and Dimitrios Tzovaras. 2025. AI-Enhanced Static Analysis: Reducing False Alarms Using Large Language Models. In *2025 IEEE International Conference on Smart Computing (SMARTCOMP)*. 288–293. doi:10.1109/SMARTCOMP65954.2025.00088

[3] Masudul Hasan Masud Bhuiyan. 2023. The Call Graph Chronicles: Unleashing the Power Within. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 2210–2212. doi:10.1145/3611643.3617854

[4] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. 169–190.

[5] 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*. 243–262.

[6] Sooyoung Cha, Sehun Jeong, and Hakjoo Oh. 2016. Learning a Strategy for Choosing Widening Thresholds from a Large Codebase. In *Programming Languages and Systems*, Atsushi Igarashi (Ed.). Springer International Publishing, Cham, 25–41.

[7] Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. 2017. Automatically generating features for learning program analysis heuristics for C-like languages. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 101 (Oct. 2017), 25 pages. doi:10.1145/3133925

[8] Peng Chen, Yuxuan Xie, Yunlong Lyu, Yuxiao Wang, and Hao Chen. 2023. Hopper: Interpretative Fuzzing for Libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1600–1614. doi:10.1145/3576915.3616610

[9] Code Intelligence. 2025. Jazzer: Coverage-guided, in-process fuzzer for the JVM. https://github.com/CodeIntelligenceTesting/jazzer. Accessed: 2025-07-17.

[10] JB Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. 2017. Xcorpus–an executable corpus of java programs. (2017).

[11] Jens Dietrich, Li Sui, Shawn Rasheed, and Amjed Tahir. 2017. On the construction of soundness oracles. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis* (Barcelona, Spain) *(SOAP 2017)*. Association for Computing Machinery, New York, NY, USA, 37–42. doi:10.1145/3088515.3088520

[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. doi:10.18653/v1/2020.findings-emnlp.139

[13] Find Security Bugs. [n. d.]. Find Security Bugs, Version 1.4.6. http://find-sec-bugs.github.io. Accessed October 2, 2018.

[14] Zhaoqiang Guo, Tingting Tan, Shiran Liu, Xutong Liu, Wei Lai, Yibiao Yang, Yanhui Li, Lin Chen, Wei Dong, and Yuming Zhou. 2023. Mitigating False Positive Static Analysis Warnings: Progress, Challenges, and Opportunities. *IEEE Transactions on Software Engineering* 49, 12 (2023), 5154–5188. doi:10.1109/TSE.2023.3329667

[15] Dominik Helm, Sven Keidel, Anemone Kampkötter, Johannes Düsing, Tobias Roth, Ben Hermann, and Mira Mezini. 2024. Total Recall? How Good Are Static Call Graphs Really?. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 112–123. doi:10.1145/3650212.3652114

[16] Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. 2020. Modular collaborative program analysis in OPAL. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 184–196. doi:10.1145/3368089.3409765

[17] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Machine-Learning-Guided Selectively Unsound Static Analysis. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 519–529. doi:10.1109/ICSE.2017.54

[18] Tin Kam Ho. 1995. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, Vol. 1. IEEE, 278–282.

[19] IBM. 2015. T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net. Accessed: 2025-07-07.

[20] Minseok Jeon, Sehun Jeong, Sungdeok Cha, and Hakjoo Oh. 2019. A Machine-Learning Algorithm with Disjunctive Model for Data-Driven Program Analysis. *ACM Trans. Program. Lang. Syst.* 41, 2, Article 13 (June 2019), 41 pages. doi:10.1145/3293607

[21] Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and scalable points-to analysis via data-driven context tunneling. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 140 (Oct. 2018), 29 pages. doi:10.1145/3276510

[22] 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. doi:10.1145/3428247

[23] 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. doi:10.1145/3133924

[24] Christian Gram Kalhauge and Jens Palsberg. 2018. Sound deadlock prediction. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 146 (Oct. 2018), 29 pages. doi:10.1145/3276516

[25] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. 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, 423–434. doi:10.1145/2491956.2462191

[26] Anant Kharkar, Roshanak Zilouchian Moghaddam, Matthew Jin, Xiaoyu Liu, Xin Shi, Colin Clement, and Neel Sundaresan. 2022. Learning to reduce false positives in analytic bug detectors. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1307–1316. doi:10.1145/3510003.3510153

[27] Ali Khatami and Andy Zaidman. 2024. State-of-the-practice in quality assurance in Java-based open source software development. *Software: Practice and Experience* 54, 8 (2024), 1408–1446.

[28] Ugur Koc, Parsa Saadatpanah, Jeffrey S. Foster, and Adam A. Porter. 2017. Learning a classifier for false positive error reports emitted by static code analysis tools. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Barcelona, Spain) *(MAPL 2017)*. Association for Computing Machinery, New York, NY, USA, 35–42. doi:10.1145/3088525.3088675

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

[30] J. Richard Landis and Gary G. Koch. 1977. The Measurement of Observer Agreement for Categorical Data. *Biometrics* 33, 1 (1977), 159–174. http://www.jstor.org/stable/2529310

[31] Thanh Le-Cong, Hong Jin Kang, Truong Giang Nguyen, Stefanus Agus Haryono, David Lo, Xuan-Bach D. Le, and Quyet Thang Huynh. 2022. AutoPruner: transformer-based call graph pruning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 520–532. doi:10.1145/3540250.3549175

[32] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 141 (Oct. 2018), 29 pages. doi:10.1145/3276511

[33] LLVM Project. 2025. *LibFuzzer - LLVM Project*. https://llvm.org/docs/LibFuzzer.html.

[34] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2024. Prompt Fuzzing for Fuzz Driver Generation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 3793–3807. doi:10.1145/3658644.3670396

[35] Wenjie Ma, Shengyuan Yang, Tian Tan, Xiaoxing Ma, Chang Xu, and Yue Li. 2023. Context Sensitivity without Contexts: A Cut-Shortcut Approach to Fast and Precise Pointer Analysis. *Proc. ACM Program. Lang.* 7, PLDI, Article 128 (June 2023), 26 pages. doi:10.1145/3591242

[36] Leland McInnes, John Healy, and Steve Astels. 2017. hdbscan: Hierarchical density based clustering. *Journal of Open Source Software* 2, 11 (2017), 205. doi:10.21105/joss.00205

[37] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Roma, Italy) *(ISSTA '02)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/566172.566174

[38] Amir M. Mir, Mehdi Keshani, and Sebastian Proksch. 2024. On the Effectiveness of Machine Learning-based Call Graph Pruning: An Empirical Study. In *Proceedings of the 21st International Conference on Mining Software Repositories* (Lisbon, Portugal) *(MSR '24)*. Association for Computing Machinery, New York, NY, USA, 457–468. doi:10.1145/3643991.3644897

[39] Roweida Mohammed, Jumanah Rawashdeh, and Malak Abdullah. 2020. Machine Learning with Oversampling and Undersampling Techniques: Overview Study

and Experimental Results. In *2020 11th International Conference on Information and Communication Systems (ICICS)*. 243–248. doi:10.1109/ICICS49469.2020.239556

[40] Austin Mordahl, Dakota Soles, Miao Miao, Zenong Zhang, and Shiyi Wei. 2023. ECSTATIC: Automatic Configuration-Aware Testing and Debugging of Static Analysis Tools. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1479–1482. doi:10.1145/3597926.3604918

[41] Austin Mordahl and Shiyi Wei. 2021. The impact of tool configuration spaces on the evaluation of configurable taint analysis for Android. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 466–477. doi:10.1145/3460319.3464823

[42] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. 2015. Learning a strategy for adapting a program analysis via bayesian optimisation. 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, 572–588. doi:10.1145/2814270.2814309

[43] OWASP Foundation. 2014. The OWASP Benchmark for Security Automation, Version 1.1. https://www.owasp.org/index.php/Benchmark. Accessed January 4, 2018.

[44] 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. doi:10.1145/3236454.3236501

[45] M Pnueli and Micha Sharir. 1981. Two approaches to interprocedural data flow analysis. *Program flow analysis: theory and applications* (1981), 189–234.

[46] Mohammad Rafieian, Vlad Birsan, Kunal Katyar, Dylan Zhong, and Shiyi Wei. 2025. Artifact for: Is Call Graph Pruning Really Effective? An Empirical Re-evaluation. https://doi.org/10.5281/zenodo.16107161.

[47] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. 2019. Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 251–261. doi:10.1145/3293882.3330555

[48] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. 2018. Systematic evaluation of the unsoundness of call graph construction algorithms for Java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops* (Amsterdam, Netherlands) *(ISSTA '18)*. Association for Computing Machinery, New York, NY, USA, 107–112. doi:10.1145/3236454.3236503

[49] Jordan Samhi, René Just, Tegawendé F. Bissyandé, Michael D. Ernst, and Jacques Klein. 2024. Call Graph Soundness in Android Static Analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 945–957. doi:10.1145/3650212.3680333

[50] Ravindra Singh, Naurang Singh Mangat, Ravindra Singh, and Naurang Singh Mangat. 1996. Stratified sampling. *Elements of survey sampling* (1996), 102–144.

[51] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 17–30. doi:10.1145/1926385.1926390

[52] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 485–495. doi:10.1145/2594291.2594320

[53] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. 2018. On the Soundness of Call Graph Construction in the Presence of Dynamic Language Features - A Benchmark and Tool Evaluation. In *Programming Languages and Systems*, Sukyoung Ryu (Ed.). Springer International Publishing, Cham, 69–88.

[54] Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 147 (Oct. 2021), 27 pages. doi:10.1145/3485524

[55] Flavio Toffalini, Nicolas Badoux, Zurab Tsinadze, and Mathias Payer. 2025. Liberating Libraries through Automated Fuzz Driver Generation: Striking a Balance without Consumer Code. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE095 (June 2025), 23 pages. doi:10.1145/3729365

[56] Akshay Utture, Shuyang Liu, Christian Gram Kalhauge, and Jens Palsberg. 2022. Striking a balance: pruning false-positives from static call graphs. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2043–2055. doi:10.1145/3510003.3510166

[57] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[58] Shiyi Wei and Barbara G. Ryder. 2015. Adaptive Context-sensitive Analysis for JavaScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs, Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 712–734. doi:10.4230/LIPICS.ECOOP.2015.712

[59] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, and Wenbin Zhang. 2021. Semi-supervised Log-based Anomaly Detection via Probabilistic Label Estimation. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1448–1460. doi:10.1109/ICSE43902.2021.00130

[60] Yixin Yang, Ming Wen, Xiang Gao, Yuting Zhang, and Hailong Sun. 2024. Reducing False Positives of Static Bug Detectors Through Code Representation Learning. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 681–692. doi:10.1109/SANER60148.2024.00075

[61] Sai Yerramreddy, Austin Mordahl, Ugur Koc, Shiyi Wei, Jeffrey S. Foster, Marine Carpuat, and Adam A. Porter. 2023. An empirical assessment of machine learning approaches for triaging reports of static analysis tools. *Empirical Software Engineering* 28, 1 (2023), 28. doi:10.1007/s10664-022-10253-z

[62] Ya-Lin Zhang, Longfei Li, Jun Zhou, Xiaolong Li, Yujiang Liu, Yuanchao Zhang, and Zhi-Hua Zhou. 2017. POSTER: A PU Learning based System for Potential Malicious URL Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2599–2601. doi:10.1145/3133956.3138825