2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P) | 978-1-6654-6512-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/EuroSP57164.2023.00037

Code Vulnerability Detection via Signal-Aware Learning

Sahil Suneja
IBM Research
Yorktown Heights, NY, USA
suneja@us.ibm.com

Yufan Zhuang
University of California San Diego
La Jolla, CA, USA
y5zhuang@ucsd.edu

Yunhui Zheng
IBM Research
Yorktown Heights, NY, USA
zhengyu@us.ibm.com

Jim Laredo
IBM Research
Yorktown Heights, NY, USA
laredoj@us.ibm.com

Alessandro Morari
IBM Research
Yorktown Heights, NY, USA
amorari@us.ibm.com

Udayan Khurana
IBM Research
Yorktown Heights, NY, USA
ukhurana@us.ibm.com

***Abstract*—Machine Learning-based modeling of source code understanding tasks has been gaining popularity. Accompanying their rapid proliferation is an emerging scrutiny over the models' reliability. Concerns have been raised regarding the models not actually learning task-relevant source code features, but fitting other correlated data. To improve model trustworthiness, in this work, we explore data-driven approaches for enhancing model *signal awareness*, i.e., learning the relevant signals in the input for making predictions. We do so by incorporating the notion of code complexity during model training, both (i) explicitly via curriculum learning, and (ii) implicitly by augmenting the training dataset with simplified signal-preserving programs. With our techniques, we achieve up to 4.8x improvement in signal awareness of vulnerability detection models. Using the notion of code complexity, we present a novel interpretation of the model learning behaviour from the perspective of the dataset. We use it to introspect model learning difficulties, and analyze the learning enhancements achieved with our approaches.**

1. Introduction

Over the past few years, Machine Learning (ML) models have made significant progress in source code understanding tasks [1], [2], [3], [4], [5]. The wide availability of open source codebases has fueled this progress, and we have started to see the adoption of such ML models of code in software development workflows [6], [7], [8]. Their growing popularity has permeated the security space as well, including source code bug detection. What used to be a domain traditionally dominated by static and dynamic analysis is seeing assistance and competition from ML models [9], [10], [11], [12], [13], [14] The high false positives of static analyzers, and the lack of completeness of dynamic analysis are a few reasons promoting the entry of ML into this field [15], [16], [17].

However, unlike the rules and path/flow analysis of static analyzers and the execution tracing of dynamic analysis, the logic learned by the ML models for detecting code vulnerabilities remains a black-box, and an emerging point of concern. Recent observations highlight that many ML models of code, despite their high F1 and accuracy scores, are actually not learning task-relevant source code features [18], [19], [20]. Instead, they often fit non-relevant but correlated data, leading to a lack of robustness and generalizability, and limiting the subsequent practical use of such models. This can happen when the models pick up non-representative code features to the task at hand, such as unexpected correlations between samples and certain keywords, or programming constructs like ifs / loops / variable names, which may be more prevalent in one sample class than the other (e.g. 'buggy' or 'healthy' code). Learning class separators in this manner may yield great classification performance. However, the models influenced by such dataset nuances– spurious correlations, sampling bias, labeling and other artifacts [21], [22], [23], [24], are prone to failures when applied in real-world settings [25], [26], [27].

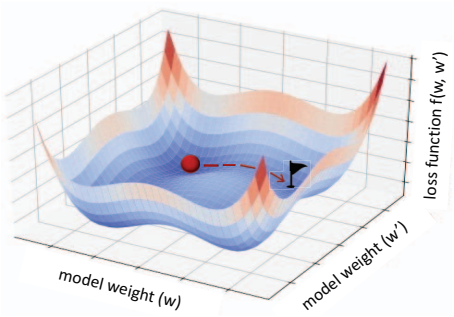


Figure 1: The intuition behind our model learning enhancement efforts is to assist the model in reaching an alternate minima (the flag in the Figure) in the loss landscape, while maintaining similar performance as achieved in original minima (the ball in the Figure), but potentially capturing more task-relevant features.

Building on top of the recent works calling for model sanity checking, in this work we focus on improving model reliability and trustworthines. Note that we are not arguing in favor of ML models versus traditional code analyzers. Although the current model proliferation trend is based on such comparison, it values statistical performance measures (e.g., F1) over model learning quality (e.g., source code features relevance). We believe both– ML and traditional approaches– can work in concert to assist with code vulnerability detection. The learned heuristics from a ML-based macroscopic approach

DOI 10.1109/EuroSP57164.2023.00037

to vulnerability detection can augment the microscopic approach of static analyzers. But to achieve that, we first need to ensure that the models are learning the task-relevant source code features in making their predictions. We work towards achieving this by targeting Software Engineering and Machine Learning collaboration, rather than competition. In addition to adding trust in ML models, this enables a more fair comparison between ML models, and also with traditional code analyzers. Our hope is to influence the future model development trend to focus more towards model learning quality, as opposed to raw classification performance.

To improve model reliability and trustworthiness, in this work, we explore the option of nudging the training process to learn models with greater *signal awareness*. Signal awareness is defined as the ability of a model in learning the relevant signals in the input for making predictions [18], as opposed to, e.g., capturing spurious correlations [21]. Specifically, the model itself may be doing its job well, learning how best to separate samples based on available input features, to reach a local minima in the loss landscape. But, as shown in Figure 1, there can exist other local minima in this landscape which can offer similar model performance, but which rely on features more in sync with the task expectation. If we can somehow guide the model to reach an alternate minima, while using domain-specific assistance, perhaps the model's signal awareness can be improved. This issue is widely seen in Machine Learning and is typically solved by white-box or domain-specific approaches such as robust training and adversarial training [28], [29], [19], [30]. In this work, we explore code-centric, data-driven approaches to guide the models in focusing more on task-relevant aspects of source code.

We observe that not all code snippets are the same; some are more 'noisy' than others– containing code not directly relevant to the learning task at hand. We target reducing this noise whilst preserving task-relevant signals. By transforming the training data using Software Engineering (SE) techniques, we aim to learn models which not only exhibit high accuracy but are also based on features relevant to the given task. We use the concept of code complexity to distinguish the different training samples, and explore ways to introduce this code-complexity awareness in the models.

Furthermore, while existing approaches can detect *if* the models are learning task-relevant signals, they do not identify *what* aspects of source code are the models learning. This can be a valuable resource towards uncovering the logic learned by models, and assert trust in them as they integrate with software development workflows. To this end, continuing along the code complexity dimension, we present a novel approach to deduce model learning from the perspective of the dataset.

Figure 2 presents a view of all our approaches co-existing within the model training pipeline, described as follows. We approach model signal-awareness enhancement by marrying the SE concept of code complexity with the ML technique of curriculum learning [31]. Specifically, we introduce the notion of complexity metrics during training, and feed program samples to the model in increasing order of their code complexity. The intuition is that by presenting 'easier' examples first, it would

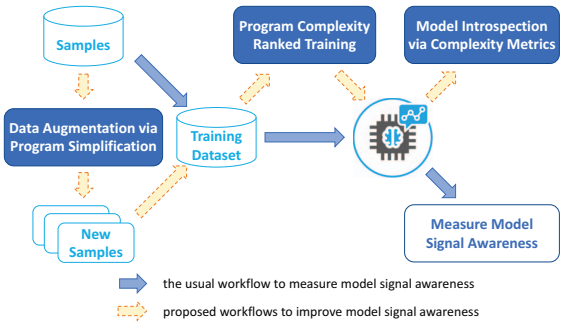


Figure 2: Our model signal-awareness enhancement (complexity-ranking and program-simplification) and introspection approaches, augmenting the usual model training workflow.

improve the model's chances of picking up task-relevant signals, helping it to sift-through noise with more complex examples down the line.

Next, we present an alternate data-augmentation-based approach to assist model learning, with the notion of complexity of code being implicit, in contrast to the previous approach. Here, we incorporate SE assistance into ML model training by customizing Delta Debugging [32] to generate simplified program samples. The intuition is that by adding smaller and potentially de-noised code samples to the training dataset, while preserving their vulnerability profile, the model can learn relevant signals better. Within the vulnerability detection setting, in particular, preserving existing bugs while generating simplified programs is the key difference in our approach. This is in contrast to existing source-level bug-seeding-based augmentation methods which can lead to previously unseen bugs [33], [34], [35], [36], [37], [38], [39].

We continue our data-driven model exploration to visualize how the altered model training process might have improved its signal awareness. We leverage the unique opportunity afforded by the source code setting to develop a code-complexity-driven black-box model introspection approach. Specifically, we analyze a model's predictions from the dataset perspective, beyond the statistical measures of the model prediction accuracy. The intuition is to analyze the characteristics of the samples which the model predicted correctly versus the mispredicted ones. By using code complexity metrics to group samples by prediction accuracy, our approach uncovers model learning behavior in terms of the aspects of code the model can grasp, and where it faces difficulties. More concretely, using this approach, we are able to ascertain that the models are correctly predicting smaller and low-complexity samples (using `sloc` and `cyclomatic` complexity metrics-Section 3.1), but facing difficulty predicting bigger and more complex code samples. We believe that such code-centric model learning insights offered by our approach are more developer friendly, as compared to the black-box quantitative measures of model performance.

Being data-driven, all our approaches are independent of the model learning algorithm, the learning task, and the source code programming language employed. Results show substantial improvements in a model's signal awareness when assisted with our approaches, across dif-

ferent datasets and models. Complexity-ranked training can provide up to 32% boost to the model's Signal-Aware F1 (Section 4.2) whereas program-simplification-based augmentation surpasses it, realizing up to 4.8x improvement. We use our data-driven model introspection approach to analyze these model learning improvements achieved. Amongst the model learning insights provided by our approach, in addition to the aforementioned issue of the models facing difficulty in understanding bigger (and more complex) code samples, our approach suggests that this problem interestingly alleviates as augmentation levels increase.

The main contributions of our work are as follows:

- We improve the signal awareness of ML vulnerability detection models, using SE concepts to assist ML understanding of source code.
- We tailor ML's curriculum learning for the source code domain, by coupling it with the notion of code complexity.
- We show the superiority of targeted augmentation with simplified programs, over generic augmentation, to improve model learning.
- We present a novel perspective for deducing model learning behavior using code complexity of the dataset.

2. Background

We first summarize the friction between traditional analysis and Machine Learning (ML) approaches. Then, we describe the ML models of code we evaluate our learning enhancement approaches against. Next, we present a brief primer on the Delta Debugging technique, and it's recent use in ML model probing. We employ it for our program simplification approach. Finally, we briefly describe the concept of Curriculum Learning, which we use in our complexity-ranked training approach.

2.1. Traditional vs. ML approaches

ML approaches for vulnerability detection cite some known shortcomings of traditional analysis to make their case. These include the potential of False Positives (flagging healthy code as being buggy) in static analysis (e.g., Clang [40]), as well as lack of completeness in dynamic analysis (e.g., Pin [41]). This happens due to the vast search space in case of a static analyzer's reasoning over all possible program states, which necessitates employing information-losing abstractions during the analysis. In case of rule-based linters and taint analysis, the quality of the rule-based static analyses depends on the quality and bug coverage of the rules. On the other hand, generating specific inputs to drive execution to relevant regions (e.g., potential vulnerability), as in dynamic analysis (e.g, fuzzing or symbolic execution), can be challenging in terms of achieving good code coverage or satisfying particular path conditions.

ML approaches try to alleviate these issues by learning code-to-vulnerability mapping heuristics automatically. As opposed to classical ML, which requires converting code into explicit features, such as number of lines, call-stack depth, library-calls, and code complexity, etc., deep learning models can extract code features automatically. With their success in the natural language processing (NLP) tasks, combined with the language-like 'naturalness' of source code [42], neural network based models have made significant inroads in the vulnerability detection task. By being exposed (trained) to enough examples (and counter-examples), the model can automatically learn *relevant* code features or vulnerability templates [43], to be able to differentiate between buggy and healthy code.

ML models of code have been shown to outperform static analyzers in recent work [44], [9]. However, ML approaches are not free of flaws. Unlike the rules and flow analysis of static analyzers and the execution tracing of dynamic analysis, the logic learned by the ML models for detecting code vulnerabilities remains a black-box, and an emerging point of concern. Recent observations highlight that many ML models of code, despite their high F1 and accuracy scores, are actually not learning task-relevant source code features [18], [19], [20]. Instead their supposedly high classification performance is derived in part by learning spurious correlations, which can happen owing to dataset nuances, sampling bias, labeling and other artifacts [21], [22], [23], [24].

We believe a collaboration, instead of competition, between ML and traditional approaches can lead to improved code vulnerability detection. Thus, in this work, we build on top of the recent works calling for model sanity checking, and focus on improving model reliability and trustworthines.

2.2. ML Models of Code

Follwing are three neural network architectures which have been popularly employed for learning over source code, each operating at a different code representation. We evaluate our learning enhancement approaches atop these for the vulnerability detection task.

Convolutional neural networks (CNNs) learn on image inputs, and have served as a fundamental tool in computer vision, particularly in image recognition and object detection [45], [46], [47], [48]. CNNs and their variants have also found utility in the domain of source code analysis [3], [49], particularly in the area of vulnerability detection [9]. In this context, source code tokens are first projected into an embedding space that is later fed into a CNN, like an image. A typical CNN is composed of convolutional and pooling layers. The former act as filters that extract features from inputs, learning progressively more intricate patterns as the neural network becomes deeper. Pooling layers, on the other hand, downsample the features in order to enhance the signal and regulate the neural network's size. Pooling is accomplished either by selecting the most strongly activated neurons (i.e., max-pooling) or by taking their average (i.e., mean-pooling). Consequently, the convolutional layers learn abstract representations of source code tokens while the pooling layers filter out extraneous inputs, yielding the source code regions most discriminative of the samples belonging to different classes (buggy vs. healthy). For example, learning the correlation between the token region around memory allocation or a size parameter, to an overflow bug.

Recurrent neural networks (RNNs) are specifically constructed to learn from sequential inputs, such as text and audio [50], [51], and have been adapted to source code vulnerability detection by processing the input program as a sequence of tokens [9], [10], [12]. Within an RNN, working memory is maintained and modified by a sequence of input, output, and forget gates, using both the current input and the previous memory state at each step [52], [53]. RNNs allow for sequential token dependencies to be captured; for example, tracking the flow of a token across the program statements, and learning what flow sequences (and what portions within them) are indicative of (ab)normal program behavior, based upon the corresponding samples' class labels.

Graph neural networks (GNNs) have gained increasing popularity due to their unique ability to learn over graph-structured data, such as social network graphs and molecular structures [54], [55]. Applying GNNs to source code is a natural fit since various forms of graphs can be constructed on top of source code, such as abstract syntax trees (AST), data flow graphs, and control flow graphs. Most GNNs consist of three modules: (i) message passing, which determines how information is exchanged among nodes via edges, (ii) message aggregation, which determines how each node combines the received messages, and (iii) message updating, which controls how each node updates its representation after one cycle of information propagation [55], [56], [57]. GNNs have achieved state-of-the-art performance on several software engineering tasks, including vulnerability detection [11], [58], [59]. When source code is represented as a graph, the nodes represent string tokens (or intermediate representation labels such as those of an AST, etc.), and edges represent their relationships (e.g., use-define). A vulnerability can be defined as a specific template of relationships between the different nodes and edges, to be learned during training. For example, the absence of a 'variable sanitization template' [43], i.e. value-range validation prior to being used as a memory allocation size argument, may indicate a potential buffer overflow vulnerability.

Depending upon the composition of the training datasets (Section 4.1), the models can be trained to detect different kinds of software vulnerabilities, including buffer overflows, integer overflows, resource leaks, deadlocks, null pointer dereference, and OS injection, amongst others [60]. The goal with our learning enhancement approaches is to assist the models in picking code features relevant to vulnerabilities, while learning their vulnerability detection logic, as opposed to capturing spurious correlations.

2.3. Delta Debugging

Delta Debugging (DD) was first proposed to minimize long crash-inducing bug reports for Mozilla's web browser. The DD algorithm iteratively reduces an input sequence of instructions (e.g., the bug report) to a minimal snippet (called *1-minimal*), while preserving the original outcome (e.g., browser crash). It involves the following sequence of operations, illustrated in Figure 3: (i) The input sequence is split into N segments (starting at N=2), and their complements. All splits are tested against an oracle function to checks if any of them lead to the same

#	Program	Valid	Vul
1	`void foo (int a) {int b = 10; int buf[10]; a + 3; buf[b] = 1;}`	✓	✓
2	`10]; a + 3; buf[b] = 1;}`	✗	
3	`void foo (int a) {int b = 10; int buf[`	✗	
4	`int b = 10; int buf[10]; a + 3; buf[b] = 1;}`	✗	
5	`void foo (int a) {  [10]; a + 3; buf[b] = 1;}`	✗	
	.....		
13	`void foo (int a) {int b = 10; int buf[10];  buf[b] = 1;}`	✓	✓
	.....		
21	`void foo (int a) {int b = 10; int buf[10];  buf[  = 1;}`	✗	
22	`void foo (int a) {int b = 10; int buf[10];  buf[b]  ;}`	✓	✓
	.....		

Figure 3: Generating signal-preserving simplified programs with our program simplification approach. Iteration #1 shows a valid but vulnerable program. In #2-3, DD first cuts it into half but fails to find a valid sub-program. DD repeatedly tries a finer granularity until reaching a single-token level. It finds two valid and vulnerable subprograms in #13 and #22. Further reduction possible but omitted.

outcome as the original input. (ii) If the test result of a segment is the same as that of the original input, it is treated as the sequence for the next iteration and the split granularity is reset. Otherwise, if the complement split has the original outcome test result, it becomes the exploration candidate for the next iteration. (iii) If none of the segments preserve the original outcome during testing, N is doubled to split the sequence into smaller segments. In each round, DD tries to reduce the scope to a subset, till not a single element can be removed whilst preserving the original outcome, thereby yielding the *1-minimal*. For more details, please refer [32].

2.3.1. ML model probing with DD.

Recent works [20], [18] use DD for probing ML models of code to explain them, or expose a lack of signal awareness. Given a trained model and an input program correctly predicted by the model, DD is used to extract the minimal sub-program (1-minimal) which preserves the model's prediction, by iteratively reducing the program and querying the model on it. This represents the minimal excerpt of the original program which the model requires to arrive and maintain its original prediction. Finally, the model's signal-awareness is determined by testing the 1-minimal (which the model predicts to be vulnerable) for the original vulnerability existence. Widespread occurrence of cases as shown in Figure 4 expose the lack of signal awareness in today's ML vulnerability detection models, despite their high F1 and accuracy scores.

Model probing customizes DD's test oracle function by employing the Infer tool [62] to verify bug existence in the reduced subprograms. At each iteration of the reduction cycle, Infer's analysis of the reduced subprogram is compared with that of the original program sample, to ensure that the reduced subprogram is either bug-free, or possesses only the same bug as the original program sample. The latter is detected by a hit for the original bug in Infer's `preexisting.json` and a miss in `introduced.json` differential analysis comparison files.

Although the existence of a perfect bug checker precludes the need for ML for code analysis, yet the latter

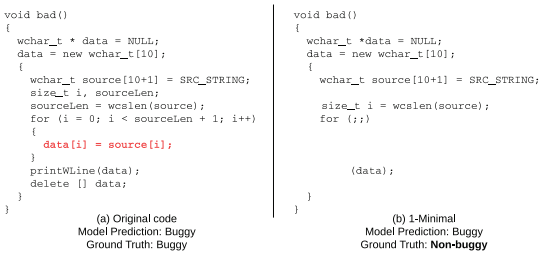


Figure 4: Example showing lack of signal awareness in a vulnerability detection model (Model: CNN, Dataset: Juliet, F1: 97). Comparing original code vs. it's extracted 1-minimal. The model predicts both as being buggy even when the 1-minimal doesn't contain the actual buffer overflow bug of the original parent code (shown in red). This suggests that the model learns signals not relevant to the task. Note that the model is trained on "regular" C/C++ examples of healthy and buggy code from the Juliet dataset [61], and not "artificial" looking code as in the right half of the figure. The latter is what is extracted using the model-probing approach of [18], and shows the actual "focus areas" of the trained model.

proliferates unvetted (Section 1). To help with model quality control, model probing approaches aim to incorporate such SE-tool-driven sanity check into ML model learning. Furthermore, although Infer as a bug checker is a fortunate fit for the vulnerability detection task, like other static analysis tools it is not perfect. But it is at least more strongly vetted and based on traditional hardened static checking principles, being used in popular open source projects.

2.3.2. Program simplification with DD.

We also adapt DD for program simplification, translating the DD process of continuously reducing the input while maintaining the same output, into successively simplifying the input program while maintaining its vulnerability profile. Each valid DD iteration generates code simpler than the parent, which we augment to the original dataset for subsequent model training. *Note that there is no measurement bias in the previous use of DD for signal-awareness measurement [18], and its use in our program simplification approach.* Unlike signal-awareness measurement, program simplification does not include the model in the DD reduction process. While the former preserves the original prediction of the model-under-test while reducing an input program, and only then tests whether or not the original bug is present in the 1-minimal, our program-simplification approach preserves the bug while successively simplifying an input program, independent of any model. Model signal-awareness gets tested as before on the original test-set samples, not on any new simplified samples. The reduced programs used to query the model during signal probing are not 'leaked' to the augmented training dataset.

Finally, while we use DD to reduce and simplify program samples, our approach are not reliant on it. DD offers an efficient reduction solution and can be substituted by other alternatives along the efficiency vs. simplicity spectrum (Section 6.3). Also, in our experiments, we use the Infer tool to verify bug existence in the reduced samples. However, our approach is independent of the specific bug-checker being employed (alternatives in Section 6.2).

2.4. Curriculum Learning

Curriculum learning [63] is a training strategy that mimics the way humans learn, by gradually introducing easier tasks before tackling more complex ones. The basic idea behind curriculum learning is to use a sequence of tasks, ordered by increasing level of difficulty, to train a model. By doing so, the model can learn more effectively and achieve better performance than training on a random task order. One key advantage of curriculum learning is that it can help prevent the model from getting stuck in suboptimal solutions, since it starts by learning simpler patterns before moving to more complex ones. Moreover, curriculum learning can reduce the amount of data needed for training since the model is gradually introduced to the complexity of the task. In this work, we tailor curriculum learning towards source code modeling, by coupling it with the notion of code complexity, as described in Section 3.1.

3. Design

Figure 2 (Section 1) presents a high-level overview of all our data-driven approaches coexisting within the typical ML modeling pipeline. Our program-simplification-based augmentation technique resides in the dataset curation phase, while our complexity-ranking approach is applicable during the model training phase. Finally, once a trained model is available, our introspection approach comes into play to deduce model learning behavior. All approaches can exist independently of each other.

3.1. Code-complexity-ranked Training

Our model training approach marries the ML concept of curriculum learning [31] with the SE notion of source code complexity. The idea is to present the learner with simpler code samples initially during training, and to increase sample complexity progressively, influenced by the human learning procedure. Coming from the original training set, these initial samples still contain the same traits which we want the model to learn, while being relatively easier than their counterparts. This can improve the model's chances of picking task-relevant signals better, with less interference from potential 'noise' existing in more-complex samples- in the form of statements or constructs not relevant to the task at hand. The intention is that equipped with the knowledge of the right 'signals' to look for, the model will be better able to sift through noise in the rest of the samples, and refine its learning while maintaining task-awareness.

Figure 5 presents our complexity-ranking training approach. The first step is the extraction of code complexity metrics from the training set samples. We used Frama-C [64] and Lizard [65] tools to extract relatively straightforward counters such as `sloc`, `ifs`, `loops`, as well as higher-order `cyclomatic` and `halstead (volume, difficulty, effort)` complexity metrics, measuring concepts such as linearly

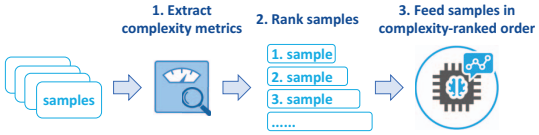


Figure 5: Our complexity-ranked training approach: 1. Extract code complexity of training samples. 2. Rank them in increasing order of complexity. 3. Feed them to model in complexity-ranked order.

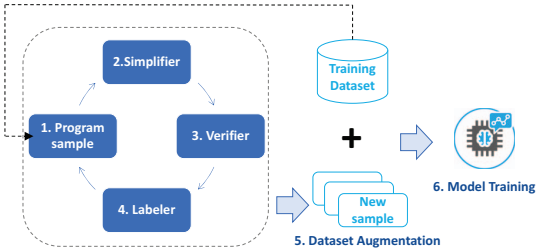


Figure 6: Our program-simplification-based augmentation approach: 1-2. For a training set sample, reduce it via DD. 3-4: Validate, label and emit reduced sample; continue reduction. 5-6: Train model on training set augmented with all simplified samples.

independent paths and coding effort required, amongst others. These commonly used code metrics serve to highlight the potential of complexity-ranked training, and can be replaced by other appropriate metrics of choice.

Next, the samples are ranked in the increasing order of their corresponding metric(s) score. Finally, the samples are then fed to the model for training in their complexity-ranked order (e.g. `difficulty` 12 $\rightarrow$ 17 in the left half of Figure 7), as opposed to generic random-sampling based training. As we show in Section 5.1, introducing code complexity awareness into model training in this manner, can improve the model's signal awareness significantly. The multiple complexity metrics options, in this setting, serve as tunable knobs to influence model learning.

3.2. Augmentation via Program Simplification

For our second model learning enhancement approach, we use a data-augmentation route with the notion of complexity of code being implicit, in contrast to the previous approach. However, instead of just offering more examples, we 'simplify' them while preserving the signals.

Figure 6 presents the overall flow. We borrow the popular fault-isolation technique of Delta Debugging (DD), shown as *Simplifier* in Figure 6, to generate simplified program samples from the training dataset. We follow the same procedure as outlined in Section 2, wherein DD reduces the program samples at the level of source code tokens. For each input sample, the intermediate subprograms generated during the reduction cycle, which satisfy certain prerequisites, are emitted to serve as augmentation candidates. Each valid iteration of the reduction cycle generates code smaller than the parent. The successive denoising achieved as a result, carries the potential of assisting the model in learning the relevant signals better,

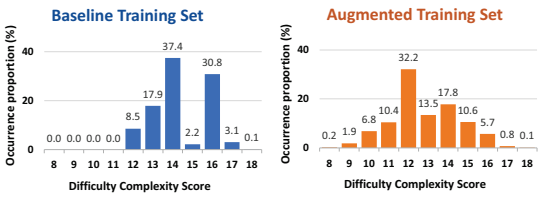


Figure 7: Complexity distribution of s-bAbI training set before and after augmentation. Note the *histogram left-shift*, signifying increased proportion of lower-complexity samples, as our simplification approach augments the training set.

when trained on the augmented dataset. Figure 7 shows the effect of our augmentation approach adding simpler examples, in terms of the dataset complexity distribution.

The iterative DD process is driven by an oracle [32], which decides whether or not an intermediate reduced subprogram should be picked for subsequent reductions. We customize the oracle with a *Verifier* and a *Labeler* to require the reduced subprograms to satisfy the following properties:

- *Valid program (Verifier).* We enforce that the reduced subprogram is valid and compilable, to ensure models aren't trained on incorrect code later on.
- *Vulnerability type (Labeler).* We additionally check the reduced subprogram for either possessing the same bug as the original sample, or being bug-free (mechanism details in Appendix A.3). By ensuring no new bug gets introduced during reduction cycle, it maintains dataset integrity, so as not to emit samples with out-of-dataset labels.

Each reduced subprogram satisfying above properties is correspondingly labeled and emitted, and the reduction cycle continues. Figure 3 illustrates an example reduction.

The overall reduction cycle results in multiple simplified samples being generated from each training sample, with each valid iteration generating code smaller than the parent. The final step is to assist model learning by adding these smaller, potentially de-noised code samples into the training mix.

3.2.1. Discussion. Use-Cases. While we use program simplification to improve signal awareness, our approach can be used as a standalone augmentation scheme to either reduce overfitting- by adding all generated samples to the training set, or to reduce class imbalance- by adding only minority-class generated samples. It can also be combined with our complexity-ranked training approach, by (i) applying the latter atop the augmented set to assist model learning (Section 5.3), or (ii) training and comparing the model on subsets ordered by program simplicity to verify model capacity and quality.

Labeler Customization. For the experiment settings and the datasets considered in this paper (Section 4.1), we found the Infer analyzer [62] to work quite well as a labeler (Infer mechanics as in Appendix A.3). However, our approach is not reliant on it– Section 6.2 discusses other alternatives.

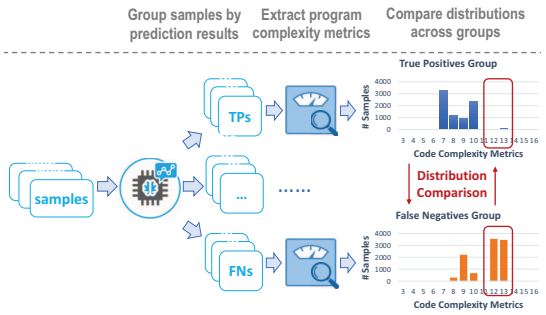


Figure 8: Our code-complexity-driven model learning introspection approach.

3.3. Model Learning Introspection

Moving beyond model learning enhancement, while still maintaining the notion of code complexity, we tackle another awareness-related aspect of ML models of code– uncovering the black-box model learning. Unlike our previous two approaches which change the model training routines, the goal here is to interpret the learning of an already trained model. Unlike the few existing model explanation approaches (Section 7), we approach the problem from the dataset's perspective. Specifically, we use the code complexity of the test set samples, coupled with the models' predictions, to deduce what aspects of code the model is able to grasp versus those where it is facing difficulties. *Note that our goal here is not to say how complexity metrics influence model training, but to use them to interpret the learning of an already trained model.*

Figure 8 shows the overall flow of our model learning introspection approach, outlined in Algorithm 1:

1) Given a trained model and the corresponding model predictions for the test set samples, we first group the samples by their prediction accuracy, as shown in steps 2-4 of the algorithm:
   - True Positive Samples (TP)- e.g., samples correctly predicted by the model as being 'buggy'
   - False Negative Samples (FN)- e.g., samples incorrectly predicted by the model as being 'healthy'.
2) Then, for each group, we generate a distribution of the samples with respect to their complexity metrics (steps 5 and 6 in the algorithm), as discussed in Section 3.1. This is shown in the Figure as a histogram of sample counts for different complexity values, ranging from 7 to 13 (generic complexity metric, for exemplification).
3) Finally, we compare and contrast the complexity metrics distributions across groups (step 7 in the algorithm), to concretely highlight the differing aspects of code grasped or missed by the model. Figure 8 shows an example where the model almost always predicts high complexity samples (with complexity values 12 and 13) incorrectly. This is indicated by the highlighted FN bars having negligible presence in the TP samples distribution.

Thus, the intuition with our introspection approach is to compare and contrast such complexity distributions, across the different test set samples grouped by their model prediction outputs. This can then highlight the nature (complexity) of the samples the model is understanding well (e.g., the TPs), versus where it is struggling (e.g., the FNs). In our experiments, we use such data-driven model prediction analysis to trace how model learning evolves across the iterations of our aforementioned program-simplification-based augmentation.

Algorithm 1 Code Complexity Driven Model Learning Introspection

Input:
M: A trained model
S: Test set corresponding to the dataset model M is trained upon
F: Filtering function for samples in test set S, based on predictions by model M

Output:
- Code-characteristics differences across filtered sample groupings

1: **function** ANALYZE(M, S, F)
2: $C \leftarrow$ EXTRACTCOMPLEXITYMETRICS(S)
3: $G \leftarrow s$ **for** s **in** S **if** MAP($F, M(s)$)
4: $G' \leftarrow S - G$
5: $H \leftarrow$ DISTRIBUTIONHISTOGRAM($C(s)$ **for** s **in** G)
6: $H' \leftarrow$ DISTRIBUTIONHISTOGRAM($C(s)$ **for** s **in** G')
7: **return** DIFF(H, H')

4. Experiment Configuration

4.1. Datasets and Models

Accurate signal-awareness measurement (as discussed in Section 2.3.1), and it's subsequent enhancement with our approaches, requires compilable code with ground-truth bug locations, beyond the class labels which vulnerability datasets are commonly limited to. Thus, for our experiments, datasets from Draper [9], Devign [11] and ReVeal [24] are excluded because they do not specify bug locations. Also, samples from VulDeePecker [10] and SySeVR [12] are slices converted into linear sequences, not valid compilable code which models are trained upon and thus excluded. The viable candidates then include two synthetic datasets- s-bAbI [44] and Juliet [61], and one real-world Github-derived dataset- D2A [66], each comprising of examples of healthy and buggy C/C++ functions. As for the remaining candidates as summarized in [24], Juliet already includes a large majority of the C/C++ subset of SARD, whilst FFMPeg+Qemu is just a publicly released subset of Devign, and NVD is a subset of the VulDeePecker samples, already discussed above.

We apply our learning enhancement approaches on three popular neural network architectures for vulnerability detection, with the learning task framed as a binary classification problem– predicting program samples as healthy (label 0) or buggy (label 1). These include: (i) a CNN learning a pictorial relationship between tokens and underlying bugs, (ii) a RNN treating code as a sequence of tokens, and (iii) a GNN operating on a graph-level representations of source code. Our model implementations are based upon the architectures presented in Russell et al. [9], VulDeePecker [10], and Devign [11].

As is commonplace, models were trained separately for the different datasets, each with its own {train, validation, test} sets. Appendix A.1 and A.2 contain details

regarding the dataset composition, model selection (e.g., Neural Networks vs. Classical Machine Learning) and descriptions, and training parameters.

4.2. Metric: Signal Aware F1

We use the methodology proposed by [18] to measure the signal awareness of the models, as well as its subsequent enhancement by our approaches. Operating atop vulnerable samples in the test set, signal-awareness measurement boils down to counting how often, when a model predicts a sample to be 'buggy' (i.e. True Positives (`TP`), rest being False Negatives (`FN`)), it uses the right signals to arrive at its prediction[1]. Using a Delta-Debugging-style minimization cycle with the model in the loop, the latter is decided by checking the 1-minimal for the presence (`TP'`) or absence (`FN'`) of the original program sample's bug (Section 2).

We extend the Signal-Aware Recall metric proposed in [18] to calculate overall model performance in terms of Signal-Aware F1 (SAF1). While Recall is defined as `TP` / (`TP` + `FN`), Signal-Aware Recall is defined as `TP'` / (`TP'` + `FN'` + `FN`) or `TP'` / (`TP` + `FN`) since `TP` = `TP'` + `FN'`. SAF1 is then calculated[2] using these signal-aware variants of Recall and Precision.

By definition, SAF1 $\leq$ F1, and the expectation is to observe a shortening of this gap in the experiments, with our signal awareness enhancement approaches. We use a relative SAF1:F1 ratio[3] (≤ 1) in our experiments, encapsulating how much of the model's performance is attributable to task-relevant signal learning. As shown in Appendix Table A.1, model performance does not get compromised in our experiments, while we strive towards improving the model's SAF1:F1. Appendix A.4 discusses details regarding baseline signal awareness measurements of the models in our experiments, and it's comparison to previous work. *Note that F1 and SAF1 for all model training configurations—baseline, complexity-ranked, as well as dataset-augmented—are evaluated on the dataset's original untouched test-set itself.*

4.3. Research Questions

Following are the main research questions we aim to explore with our experiments:

1) What impact does complexity-ranked training have on model learning behavior, and does it improve model signal awareness?
2) What impact does program simplification based augmentation have on model signal awareness, and is it better than generic augmentation?
3) What sort of model learning deduction can be obtained by leveraging the dataset's code complexity distribution, and is it more insightful than usual statistical measures?

1. Signal awareness measurement on other groups, e.g. False Positives (FP; samples incorrectly predicted by the model as being 'buggy'), is not very useful as it doesn't give any insights on whether the model is learning any vulnerability-specific signals.

2. The F1 statistical measure of performance is calculated as 2 * Precision * Recall / (Precision + Recall)

3. Mathematically, SAF1:F1 == SAR:Recall == `TP'` / `TP`.

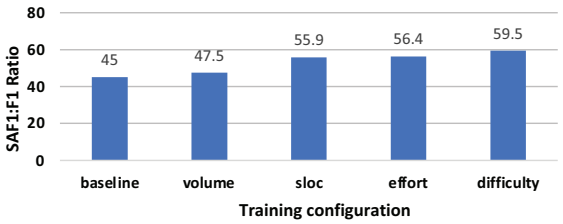


Figure 9: Model SAF1 improvements with complexity-ranked

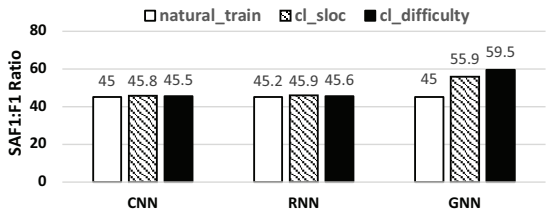


Figure 10: SAF1 improvement over baseline (natural_train) with two complexity-ranked training schemes, across models. [s-bAbI]

4.4. Baselines and Competing Approaches

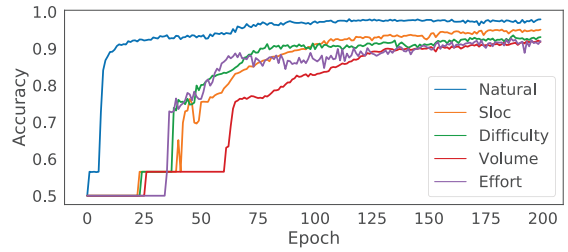
The baseline performance metrics are gathered using model implementations based upon the architectures proposed in Russell et al. [9], VulDeePecker [10], and Devign [11] [4]. With the alternative model training approaches proposed in this paper, the *competing approaches* include: (i) the usual random-sampling based training, as opposed to complexity-ranked training, and (ii) generic augmentation (adding more training samples from the dataset), as opposed to augmentation with simplified programs. These are shown as baseline configurations or leftmost bars in the Results Section graphs (Section 5), upon which we show superior model signal-awareness improvements with our approaches.

5. Results

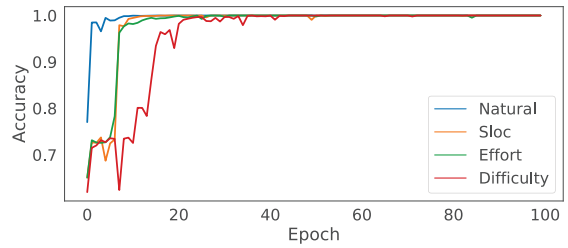
5.1. Code-complexity-ranked Training

Figure 9 shows how a model's signal awareness can be significantly improved by presenting it source code samples in the increasing order of code complexity. Shown is the SAF1:F1 ratio for a GNN model on the s-bAbI dataset for different training configurations, including natural random-sampling based training (`baseline`), and four complexity-ranked training schemes based upon `sloc`, `volume`, `difficulty`, and `effort` complexity metrics respectively. As can be seen, complexity-ranked training can significantly boost the model's signal awareness, with **`difficulty`-ordered training achieving a 32% improvement.** However, not all models show such improvements, as shown in Figure 10 comparing CNN, RNN and GNN models across a couple such complexity-ranked training schemes (same results for others; not

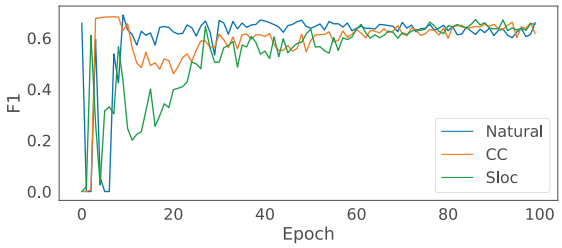
4. While these ML approaches claim to be better than traditional analyzers, this is not the claim or focus of this work.



(a) s-bAbI dataset



(b) Juliet dataset



(c) D2A dataset*

Figure 11: Comparing validation-set performance curves. Complexity-ranked training is slower than natural training, but eventually makes up. *: F1 curve for D2A for better visualization; accuracy curve noisier but has same trend. (CC = cyclomatic complexity). [GNN]

shown). The fact that other models aren't able to improve may be due to the better learning potential for a model trained over a more natural graph-based representation of source code, than over code-as-image (CNN) or code-as-token-sequence (RNN) counterparts [67], [68], [69].

Figure 11a shows how the model learning changes with complexity ranked training for the GNN model on the s-bAbI dataset. It shows the validation accuracy curves (i.e. model's interim accuracy on the validation set as it progresses along its training rounds or 'epochs') for the different training configurations– natural training, and the four complexity-ranked training schemes. As can be seen, natural training quickly reaches quite close to its peak performance, whereas the learning is relatively slow with the complexity-ranked schemes. Although all eventually reach similar looking peaks, the crucial difference is the minima reached by these training configurations. Even though both natural and complexity-ranked training schemes reach a 90%+ accuracy (and F1), the latter is much more task-aware, as shown before in Figure 9's signal-aware F1 (SAF1) values. This can be attributed to the model learning, albeit slowly, better signals from 'easier' examples first, empowering it to sift-through noise with more complex examples later.

This altered training behavior is also seen in the other models and datasets, with Figures 11b and 11c showing similar behavior for certain Juliet and D2A example configurations (similar trend with other model-metric configurations; omitted). Although complexity-ranking alters the model's learning route, and by itself can offer some assistance to the models, it doesn't seem to always help the model reach a more task-relevant minima. The baseline SAF1:F1, obtained with the natural training scheme, **remains unchanged for the Juliet dataset even with complexity-ranked training, while D2A shows only a 3.5% improvement.** But as shall be shown later (Section 5.3), complexity-ranking still has some benefits to offer in those settings.

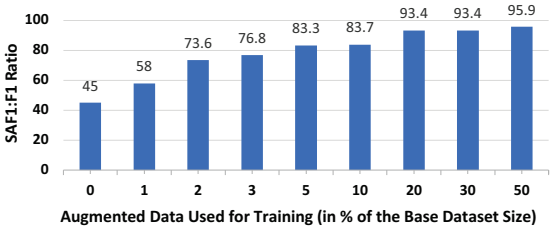


Figure 12: GNN SAF1 improvements with random sampling over the augmented s-bAbI dataset.

Summary: *The simplicity of complexity-ranked training, together with the altered model learning, offers some assistance to model signal awareness, although not universally.*

5.2. Augmentation via Program Simplification

Our program simplification approach results in the **generation of 9x more samples for s-bAbI, 9.6x for Juliet, and 53x for D2A**, as a factor of the base dataset size. The varying levels of augmentation are due to the difference in the datasets' sample sizes, which tend to be much bigger for the real-world D2A dataset, as compared to s-bAbI and Juliet (median sloc 36 vs. 9). The bigger the input code sample, the more the number of reduction iterations performed by Delta Debugging, resulting in potentially more valid intermediate samples being generated.

Training the models over these additional (and simplified) samples yields even greater signal awareness improvements than achieved with complexity-ranked training. This can be seen in Figure 12, showing SAF1:F1 values achieved with different levels of augmentation for the GNN model over the s-bAbI dataset. The x-axis shows the proportion of samples (in the percentage of the base dataset size) randomly selected (repeated and averaged) from the generated set, and added to the base dataset for training, with the leftmost point (x=0) referring to the baseline model performance (same as shown in Figure 9). As can be seen, by introducing just 2% additional simplified program samples into the training mix, the model signal awareness improvement surpasses that achieved with complexity-ranked training. The gains continue with more augmentation, with SAF1 reaching almost 96% of its attainable max (i.e. F1) with 50% augmentation, amounting to a **113% improvement over the base model signal awareness.** By presenting the model with smaller

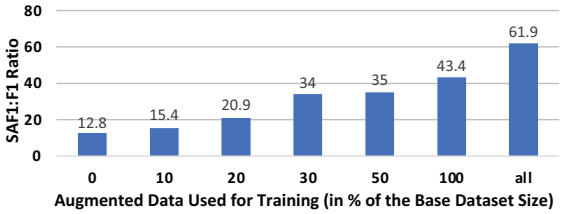


Figure 13: GNN SAF1 improvements with random sampling over the augmented Juliet dataset.

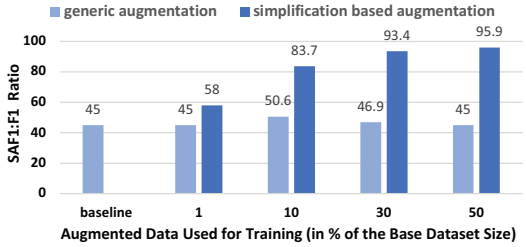


Figure 14: Comparing the generic and program simplification based augmentation approaches on the s-bAbI dataset.

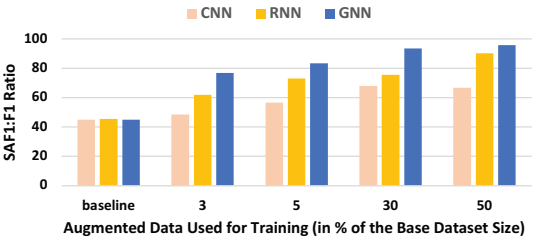


Figure 15: Program simplification based augmentation improves the SAF1 of CNN, RNN and GNN on the s-bAbI dataset.

samples, while still containing the characteristics relevant to the task at hand (i.e. bugs), it seems to be helping the model focus more on task-relevant aspects of code and less on noise or dataset nuances. More concrete insights on how model learning is changing under the covers shall be revealed in Section 5.4, highlighting the potential of our introspection approach.

The trend is the same for the Juliet dataset, with more augmentation yielding greater signal awareness improvements, as shown in Figure 13 for a GNN model. The relative gains however are even more extreme, due to the poor baseline model SAF1. SAF1 improves dramatically across the augmentation levels, crossing 60% of F1 (**a 4.8x improvement**) when *all* generated samples are additionally used for training. The usual model quality metrics (Precision, Recall, F1) maintain their similar high (90+) values in all augmentation configurations, while SAF1 increases, as shown in Table A.1.

D2A records only modest SAF1:F1 improvement of 13.3%. Interestingly, the model **Recall also gains by 22.1%** during augmentation, while the Recalls in the case of s-bAbI and Juliet datasets are stably high. This suggests the base dataset is not sufficiently large for model training. Just with derived simplified examples alone, we are able to guide the model to correctly capture more signals and thus improve both the classic model Recall and signal-aware Recall. On the other hand, this also points to the diversity of the real-world programs, and the challenge of generating sufficient simplified programs that can help models distinguish various signals from noises in such a diverse code base. Note that the sophisticated methodology behind D2A curation makes it non-trivial to collect more samples and enlarge the training set [66]. It uses differential analysis based on Github commit history, for filtering the false positives from (presumably) a state-of-the-art Infer static analyzer.

These performance gains are not just due to fact that there are extra samples to train upon. This can be seen in Figure 14 which shows the SAF1:F1 values obtained with generic augmentation, compared to our approach, for a few representative augmentation levels for the s-bAbI dataset. As can be seen, **just adding more samples to the training set does not necessarily increase the model's signal awareness, unlike our simplified program sample augmentation.** The fact that the code samples generated by our approach are smaller and potentially simpler than the original samples, is crucial to the model being able to better capture task-relevant signals and sift through noise during training. The results are the same for the Juliet dataset, with generic augmentation not improving upon the baseline SAF1 at all, irrespective of the augmentation level, very much unlike our augmentation approach. As for D2A, since it is already very limited in size, so there isn't enough hold-out data to test generic augmentation.

Unlike the case with complexity-ranked training, the signal awareness of CNN and RNN models is also given a boost with our augmentation approach. This is shown in Figure 15 showing SAF1:F1 values for the three models for a few example augmentation levels (similar trend for other model-dataset configurations; omitted). As can be seen, the GNN model outshines the competitors as was the case with complexity-ranked training, again highlighting the superior potential of code-as-graph based modeling for better learning with appropriate guidance.

Summary: *Dataset augmentation with simplified, denoised program samples assists models in learning task-relevant signals better, while maintaining model performance.*

5.3. Hybrid Training

The two approaches—complexity-ranked training and program-simplification-based augmentation—are complementary to each other, and can potentially be combined together for different use-cases. These include schemes such as (i) selecting only the more complex samples to be simplified, or (ii) ordering the augmented samples in the order of their code complexity metrics during training, or (iii) training and comparing the model separately on subsets ordered by complexity, for verifying model capacity and quality, amongst others. We experiment with one such hybrid setting to explore the potential for even more gains to be had in the model signal awareness, by

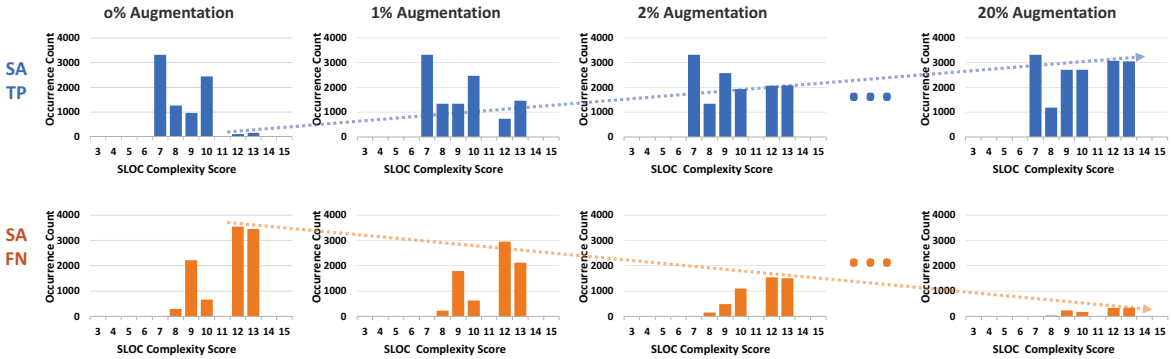


Figure 16: Insight: *More augmentation is helping model better understand bigger code samples*- via complexity distribution comparison between s-bAbI SA-TP/FN groups. "X% Augmentation" = base dataset + X% augmented samples (in % of base dataset size). Notice how the SA-TP 'skyline' (i.e. sample occurrence counts) rises with augmentation, while that of SA-FN falls. This is highlighted via the upward-trending dotted line spanning the SA-TP plots (similarly, the downward-trending dotted line for the SA-FN plots)

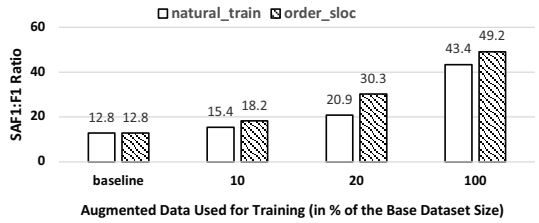


Figure 17: Hybrid training: combining complexity-ranking with program simplification based data augmentation on Juliet dataset

performing complexity-ranking atop the training dataset augmented with simplified program samples. Additional signal-awareness boost with such a combination was most evident for the Juliet dataset, as shown in Figure 17 for a few example augmentation configurations. This is in stark contrast with the inability of complexity-ranking by itself to improve model's signal awareness for Juliet (Section 5.1). One hypothesis for this is the expanded metric range which opens up post-simplification, as depicted in Figure 7, improving ranking granularity and thus new sample ordering during training.

5.4. Model Learning Introspection

So far, we have shown the potential of different data-driven approaches to improve model signal awareness. Although we have conjectured the possible reasons behind such improvements, we haven't yet probed the model black-box. In this section, we present the results of our code-complexity-driven model introspection approach to analyze model evolution across augmentation iterations.

5.4.1. Understanding Augmentation Evolution. Our introspection approach deduces model learning behavior by comparing the complexity distributions of the test-set samples, grouped by their prediction accuracy. Recall that the signal-awareness measurement results in the correctly predicted test-set samples being divided into SA-TP and SA-FN[5], depending upon whether or not the model captured the right signals to arrive at its otherwise "correct" vulnerability prediction. We compare the code complexity distribution of the SA-TP and SA-FN samples, to interpret model learning from the dataset's perspective. While Section 5.2 showed that training on datasets augmented with simplified programs improves models' signal awareness, the goal here is to go one step beyond and trace *how* model understanding of code improves with augmentation.

Figure 16 presents this comparison using `sloc` distribution across s-bAbI augmentation iterations. Comparing the leftmost pair of SA-TP vs. SA-FN `sloc` distributions reveals the first insight regarding the **baseline model (0% augmentation) facing trouble understanding bigger code samples.** This can be seen in terms of the high occurrence count of `sloc` = {12,13} samples in the SA-FN plot, with an extremely small presence in the SA-TP counterpart. Repeating the comparison across different augmentation iterations enables tracing how the model understanding of source code evolves. Specifically, the particular code-size weakness improves as augmentation increases, as can be seen with the rising SA-TP 'skyline' (i.e. sample occurrence counts), and correspondingly the falling SA-FN skyline, most evident for `sloc` = {12,13} samples. This leads to an intriguing insight about **augmentation helping the model better understand bigger code samples.** This is especially interesting because the model learning was generic– the model was not explicitly aware of code size or complexity of samples. It is only after the fact that we analyze the model's prediction performance from the perspective of the test set's code complexity, that we uncover these findings. Furthermore, its not that the base dataset did not have enough large-sized samples for the model to train upon– the base training set consists of around 17% each of `sloc` = {12,13} samples. Each augmentation iteration introduces more samples for the model to train upon, also implying more quantity of de-noised low-complexity samples (resulting from program simplification; Figure 7), gradually improving the model's chances to learn relevant signals.

5. Referred to as TP' and FN' in Section 4.2

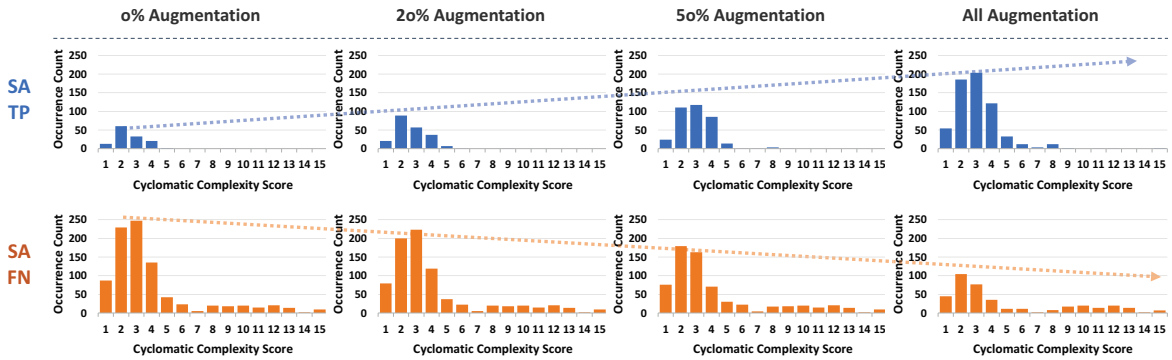


Figure 18: Insight: *More augmentation is helping model better understand more complex code samples*- via complexity distribution comparison between Juliet SA-TP/FN groups. "X% Augmentation" = base dataset + X% augmented samples (in % of base dataset size)

Different insights can be derived by changing the complexity metrics employed for model introspection. For example, Figure 18 presents a similar model evolution analysis for the Juliet dataset, but from the perspective of the `cyclomatic complexity (cc)` metric (# independent paths in the program). The same augmentation-driven 'rising / falling skyline' behavior can be seen in the `cc` distributions of TP and FN samples, revealing the model-learning dynamic about **improved understanding of code structure with more augmentation.** What does not change, however, is the occurrence counts for more complex samples (`cc` > 8) in FNs, signifying the need for potential white-box model enhancement beyond just data-driven simplified-program augmentation.

As for the D2A dataset, the modest 13.3% signal-awareness improvements recorded with augmentation, precludes observation of any meaningful evolution trends or insights during its dataset-complexity-driven analysis.

5.4.2. Understanding Augmentation-Invariant Classes.

We now use the complexity distribution comparison approach as above, to examine the characteristics of samples for which the model learning behavior remains invariant to augmentation levels. We define two categories of such samples as follows. As before, for each model M_i trained under an augmentation setting (i.e. base dataset + $X\%$ simplified samples), subsequent signal-awareness measurement results in it's true positives being divided into SA-TP$_i$ and SA-FN$_i$, depending on if M_i captured the real signals or not. Then, the two special classes in focus are: (i) AlwaysTP := $\bigcap_{i=1}^{n}$SA-TP$_i$, samples always captured correctly by the model; (ii) AlwaysFN := $\bigcap_{i=1}^{n}$SA-FN$_i$ that are consistently mispredicted. The intersection of SA-TP$_i$ or SA-FN$_i$ allows us to focus on samples that are not affected by the augmentations, and thus examine the characteristics of both- the straightforward and challenging samples- for a model architecture.

Figure 19(a) compares the `difficulty` complexity metrics distribution of the AlwaysTP samples for the s-bAbI dataset, versus the AlwaysFN group. It provides an interesting insight into the model learning behavior from the point of view of the difficulty of the program to understand (in terms operator and operand usage volune). Across all augmentation iterations, the model is more

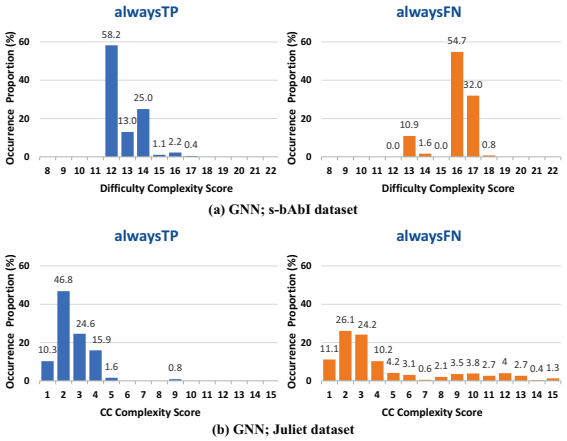


Figure 19: Comparing characteristics of samples consistently (mis)predicted by model across augmentations

easily able to **correctly capture less difficult code samples** (`difficulty` = {12,14}), while **consistently mispredicted samples tend to be harder** (`difficulty` = {16,17}). This does not mean that the latter category is never predicted correctly by the model. The model eventually learns to be able to predict them with sufficient augmentation, as we saw in the augmentation evolution Figure 16, but the ones that still remain mispredicted possess the higher metric scores.

Figure 19(b) shows the corresponding behavior for the model augmentation on the Juliet dataset, this time taking the example of the cyclomatic complexity (`cc`) metric (# independent paths in the program). The model is able to **somewhat capture low complexity samples** (`cc` < 5) as opposed to the more **complex ones (`cc` > 10), where it fails consistently**, irrespective of augmentation assistance. The separation between the distributions for AlwaysTP and AlwaysFN is not as clear as for s-bAbI, since model learning isn't as good for Juliet to begin with, with almost half of the test-set samples are consistently mispredicted for Juliet despite augmentation.

Using other metrics such as `ifs`, `loops` etc. yields more fine grained insight into model learning. For exam-

ple, for s-bAbI, across augmentation iterations, the model is more easily able to correctly capture code samples containing no loops (77%), while consistently mis-predicted samples tend to have a loop in them (88%).

Summary: *Model introspection from the dataset perspective yields code-centric, developer-friendly insights into model learning behavior, beyond the usual generic measures of model quality.*

6. Discussion and Limitations

6.1. Training Distribution Shift

During augmentation, we are adding additional data to the training set, which can be considered as sampling around the original data points[70], but in the discrete program space and with an accurate labeler. Theoretically, this may add learning difficulties for models due to the further complicated decision boundary as the data distribution shifts away from the test set. But more importantly, it would also help the model to learn real signals, therefore improving its reliability. In practice, we observed that the model performance did not deteriorate with our simplified-program augmentation, while its signal awareness increased dramatically (Table A.1).

6.2. Other Tasks, Datasets and Models

In this work, while we focus on vulnerability detection models, our approaches are independent of the target source code understanding task. The component that tailors our program-simplification approach to specific tasks is the labeler (or the task-profile checker; Section 3.2). Some options, highlighting a cost vs. quality trade-off, include: human domain expert, original dataset labeler, line-based code-feature matcher, static analyzer, and fuzzer, amongst others. The vulnerability detection setting enables leveraging readily available SE tools to ensure correctness during task-profile verification, i.e. existence of ground truth vulnerability in reduced programs.

In terms of datasets, signal-awareness measurement as per [18] requires compilable code with vulnerability location information, beyond the 0/1 labels which vulnerability datasets are commonly limited to. Datasets from Draper [9] and Devign [11] thus get excluded because they do not specify bug locations. Samples from VulDeePecker [10] and SySeVR [12] are slices converted into linear sequences, not valid compilable code which models are trained upon and are thus excluded. On the other hand, s-bAbI[44], Juliet[61] and D2A[66] do in fact contain bug-level information, and are thus considered in this work. While we were able show the effectiveness of our model learning enhancement approaches on these datasets, nevertheless, the additional ground-truth availability constraint on the datasets necessitated by signal-awareness measurements, may limit the generalizability of this paper's observations.

Appendix A.2 presents our rationale behind using neural networks as the AI model baselines in this paper, as opposed to classical machine learning approaches. While we apply our approaches on popular CNN, RNN and GNN models of source code, being data-driven and independent of the learning algorithm and the model internals, our approaches are applicable to other AI models as well. Approaches such as BERT and Transformers [71], [72], which have shown great promise in the NLP domain, have been also recently been ported to the source code domain [73], [74]. Since these are even more complex than the 'vanilla' neural network architectures, the black-box nature of their learning is even more pronounced. We plan to apply our model-agnostic approaches to analyze the signal awareness of these architectures as well.

6.3. Reduction Engine Alternatives

While we use Delta Debugging (DD) [32] to simplify program samples, our approach is not reliant on it. The core idea behind our approach remains independent of the specific program reduction engine employed– specifically, augmenting training with simpler programs while preserving signals. DD offers an efficient reduction solution and can be substituted by existing alternatives such as HDD[75], Perses[76], C-Reduce[77], amongst others. Simpler alternatives can be employed at the cost of lower efficiency, such as a linear, brute-force or randomized schemes for source code tokens/statements selection for reduction. Irrespective of the reduction scheme, the more important components are correctness validation (e.g., via a compiler) and original task-profile checking (Section 6.2).

7. Related Work

Machine learning models have garnered increasing popularity in the realm of security in recent years. Classical ML methods learn over explicit source code features such as number of lines or conditional statements, library functions, system calls, call-stack depth, complexity measures, and meta features like commit messages and bug reports. [78], [79], [80], [81], [82], [83]. Alternatively, statistical language models capture regularities in source code[84] at the token level. Specific to the bug detection task, [85] leverages N-gram language models to calculate the probability distribution of program tokens in a project, and flag low probability token sequences as potential bugs. [86] similalry uses N-gram analysis to rank the executable statements of a software by level of suspicion.

Instead of feature engineering as in classical ML, deep learning approaches automatically extract features from code by treating it as an image, a linear sequence or a graph. Specific to the bug detection task, VulDeePecker [10] trained a BiLSTM model (a RNN variant) on word2vec embeddings [87] of code snippets, surpassing several static analysis baselines. Russel et al. [88] delved further into representation learning, utilizing CNN and RNN, and bootstrapping the final prediction with the Random Forest Classical ML method. SySeVR [12] builds on this foundation by first extracting code snippets via program slicing on the program dependency graph (PDG) and control flow graph (CFG), then training embeddings via word2vec, and finally employing deep learning models, including Multilayer Perceptron (MLP), CNN, and (bi-directional) RNN, on top of the learned embeddings. GNNs have been shown to perform better than their other neural network counterparts, owing to their ability to

learn from more semantically-rich graph representation of source code. Li et al. [89] use a combination of abstract syntax tree (AST), PDG and data flow graphs to encode Java methods. Attention GRU (a RNN variant) as well as attention convolutional layer is used to focus on buggy paths in the code. Hoppity[90] adds a pointer mechanism to a GNN for bug localization, operating primarily on the AST. Unlike the general programming bugs specific to Java, as targeted by Hoppity and Li et al., Devign [11] targets exploitable C/C++ vulnerabiliies, learning over a Code Property Graph [43] representation of code (essentially AST + CFG + PDG). Allamanis et al. [42] surveys several other model variants and use-cases of ML over source code. Unlike this line of inquiry, our objective is not to develop a better model for a specific benchmark dataset, but rather to enhance the overall learning framework through signal-aware learning. As such, being data-driven, our approaches are independent of the model architecture, and can be used complementary to the model under test to enhance its signal awareness.

Despite demonstrating clear superiority over traditional static analysis methods, ML-based techniques have their own limitations and challenges. Our study is part of a group of recent works [24], [91] that uncover the potential drawbacks of ML-based methods and provide potential solutions for them. Both of these works discuss issues in the data-gathering process that could lead to dataset bias, inappropriate modeling designs that might induce the learning of spurious patterns, and improper evaluation metrics that could result in unfair comparisons. Our work complements these modeling pipeline recommendations by proposing code-complexity inspired training alternatives, to guide the model towards learning task-relevant features.

Augmentation methods are popular in AI in general, including domain-specific approaches such as image transformations [92], text transformations [93], data-driven approaches such as SMOTE [70], and formal and empirical augmentation [94], [95], amongst others. Complementary to these approaches, our augmentation approach is focused specifically towards improving the signal awareness of source code models. A key difference is that general AI approaches usually assume the input under augmentation would keep the original labels since it is extremely hard to check for images and texts without huge manual effort, while this assumption may not always be true. Our approach utilizes the benefits of working in the well-defined source code space, therefore we can assure the validity and correctness of our code augmentation. In the context of vulnerability detection, preserving existing bugs while generating simplified programs is the key difference in our approach. This is in contrast to existing source-level bug-seeding-based augmentation methods which can lead to previously unseen bugs [33], [34], [35], [36], [37], [38], [39].

Our second approach to improving model signal awareness combines code complexity with **curriculum learning** (CL) [31], [96], [97]. While CL has previously been applied using general complexity measures of images and texts to rank training samples in the vision and natural language domains, we use code-specific complexity measures to tailor the models towards source code understanding. Different to CL which is data-driven, denoising approaches [98] are usually built into the model, and assist model learning by mapping noisy input to noise-less input.

Finally, we also use the notion of code complexity to introspect model learning. Existing **explanation approaches** tend to use white-box model internals to add some transparency into the model logic. This includes probing the model's gradient [99], [100], [101], [102] to highlight input regions most influencing the model's prediction, or fitting interpretable surrogate models to approximate the deep learning model's behavior [103], [104], [105], and then using the surrogate to derive the feature importance ranking for the input. The approximate nature of such mappings, from the model side back to the data, can make them misleading [106]. Explanation methods have also been created for graphical neural networks, attributing importance to graph nodes and edges by using attention mechanisms [107], [108], or via maximizing mutual information between inputs and outputs [109]. Our model learning introspection approach is complementary to these explanation approaches as it treats models as black boxes, deducing model learning from the dataset's perspective, based on concretely defined characteristics of source code. This empowers our approach to offer more code-centric and developer-friendly insights. This is in contrast to certain other approximation-based black-box explanation approaches [20] which do not require the inputs to remain natural or valid. Furthermore, unlike our work, these do not tie source code constructs to model signal awareness.

8. Conclusion

Using SE concepts, we developed data-driven approaches to assist ML vulnerability detection models in learning task-relevant aspects of source code better. To enhance model signal-awareness, we incorporated the notion of complexity of source code into model training. We achieved significant improvements with our complexity-ranked training and program-simplification-based augmentation approaches, while maintaining model performance. We carried the notion of complexity into model introspection, and presented code-centric insights into black-box model learning. Moving forward, we look to explore active learning approaches, helping the model with targeted SE-assistance, e.g. picking specific samples for augmentation where the model is facing difficulty learning.

9. Data Availability

The datasets, source code and model checkpoints will be made available at Github, pending internal clearance.

A. Appendix

A.1. Datasets

We use the following datasets in this work:

s-bAbI: The s-bAbI synthetic dataset [44] contains syntactically-valid C programs with non-trivial control flow, focusing solely on the buffer overflow vulnerability. We used the s-bAbI generator to create a balanced dataset

of almost 40K training functions. Samples with 'UNSAFE' tag are labeled as 1, and those with 'SAFE' tag as 0.

Juliet: The Juliet Test Suite [61] contains synthetic examples with different vulnerability types, designed for testing static analyzers. From its test cases, we extract almost 32K training functions, amongst which 30% are vulnerable. Samples tagged as 'bad', and with clear bug information as per Juliet's manifest.xml file, are labeled as 1, while the ones with a 'good' tag are labeled as 0.

D2A: D2A [66] is a real-world vulnerability detection dataset built over multiple Github projects- OpenSSL, FFMpeg, HTTPD, Nginx and libtiff. It contains in-depth trace-level bug information, derived using differential analysis atop the Infer static analyzer outputs of consecutive repository versions, before and after bug-fixing commits. It comprises of a variety of vulnerabilty types including buffer overflows, integer overflows, and memory/resource leaks. Function-level sample extraction from D2A traces yields 6728 functions.

A.2. Models

We use neural networks as our ML model baselines. This is driven by two main reasons. First, existing work [9], [11] has already demonstrated the superiority of neural-network-based deep learning over classical machine learning models (e.g., random forest and XGboost) of source code. And secondly, deep learning has a more pronounced black-box nature as opposed to classical machine learning. Their ability to automatically learn input features, while enables them to learn complex functions, but it hampers their interpretability. This is in contrast to classical machine learning approaches, which can better offer visibility into their learned logic, while operating upon explicitly defined features.

We apply our learning enhancement approaches to three popular neural network architectures for vulnerability detection tasks.

A **Convolutional Neural Network (CNN)** tries to learn the pictorial relationship between tokens and underlying bugs. Similar to [Russell et al., 2018 [9]], we normalize the function names and variable names to fixed tokens such as Func and Var. We set the embedding layer dimension to 13, followed by a 2d-convolutional layer with input channel as 1, output channel as 512, and kernel size as (9, 13). The final prediction is generated by a 3-layer multilayer perceptron (MLP) with output dimensions being 64, 16, and 2.

A **Recurrent Neural Network (RNN)** treats a program as a linear sequence of tokens to learn the temporal relationship between the tokens and bugs. We implement our RNN based on [10]. We set the embedding layer dimension as 500, followed by a two-layer bi-directional GRU module with hidden size equals to 256. The final prediction is generated by a single-layer MLP. Similar to CNN, we normalize the input functions as well.

A **Graph Neural Network (GNN)** operates on the graph-level representations of source code, which are commonly used in program analysis and compilation. For example, in Devign[11], GNN tries to learn bug patterns in a Code Property Graph. We set the embedding size as 64, followed by a gated GNN layer [56] with hidden size 256 and 5 unrolling time steps. Similar to Devign, we do not normalize the tokens. The node representations are obtained via summation of all node tokens' embedding, and the graph representation read-out is constructed as a global attention layer. The final prediction is generated by a 2-layer MLP with output dimensions 256 and 2.

The models are trained over the datasets presented in Section A.1, using a 80:10:10 train:validate:test split. For all models, we set dropout rate as 0.2 during training, and used the Adam optimizer. We tuned learning rate in $\{10^{-3}, 10^{-4}\}$ and batch size in $\{24, 36, 128, 256, 512\}$. Models are trained to minimize cross entropy loss. We save the checkpoint with the least validation loss across epochs, with early stopping employed (patience = 10). Results are averages across multiple runs.

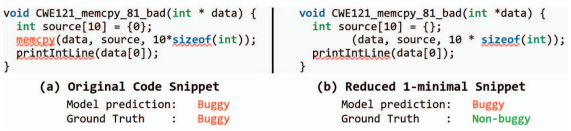


Figure A.1: A buffer overflow example showing why a looser signal-awareness measurement bound is measured by line-based bug matching. Even though the model incorrectly considers the 1-minimal as buggy, a line-based checker will count this partial match in favor of the model capturing real signals, since the buggy line exists in 1-minimal.

TABLE A.1: Using {Juliet + GNN + augmentation} example to show: (i) model performance is maintained (rows 1-3) as model signal awareness improves (SAF1 in row 4) with our augmentation approach, and (ii) improvements still achieved with our approach, even when using a looser signal-awareness measurement bound (SAF1' in row 5) (similar results for other model-dataset configurations). *Note that the very high model performance measures are in line with previously reported values [9], [110], [111], [18], owing to Juliet's synthetic nature. For comparison, for the real-world D2A dataset, the model F1 is 64.2, again comparable to previously reported values [11]*

	base + 0% aug	+10%	+20%	+30%	+50%	+100%	+all
Precision	99.9	99.9	99.9	99.9	99.9	99.9	99.9
Recall	99.9	99.9	99.9	99.8	99.8	99.8	99.9
F1	99.9	99.9	99.9	99.8	99.8	99.8	99.9
SAF1	12.8	15.4	20.9	33.9	34.9	43.3	61.8
SAF1'	49.6	51.1	54.7	61.1	61.2	65.3	74.8

A.3. Infer as a labeler

At each iteration of the reduction cycle in our program simplification approach (Section 3.2), we compare Infer's analysis of the reduced subprogram with that of the original program sample. We ensure that the reduced subprogram is either bug-free, or possesses only the same bug as the original program sample. The latter is detected by a hit for the original bug in Infer's preexisting.json and a miss in introduced.json differential analysis comparison files.

A.4. Selecting Model Signal Awareness Baseline for Enhancement

Previous work for measuring signal awareness [18] uses a combination of checkers to test bug existence in a sample's 1-minimal: utilizing the Infer analyzer [62] (checks existence of original bug), with fallback to line-based bug matching (checks existence of buggy line) for samples with differing Infer verdict and the original bug. As compared to Infer, line-based bug matching is less accurate, e.g. it counts even partial matches in favor of the model as shown in Figure A.1, thereby providing a looser signal-awareness measurement bound than Infer analysis. Despite this partial matching benefit, a lack of signal awareness in the models is still observed, as shown in [18]. However, to show the true impact of signal awareness improvements with our model learning assistance techniques, which line-based matching would mask, we instead employ the stricter Infer-based matching alone. For correctness, we focus only on the samples where Infer verdict matches the original bug. As a result of using the tighter signal-awareness measurement bound, the models' baseline performance are lower than those reported in [18]. Using a stricter checker (limited to the applicable test-set subset) thus reveals that the issue of the models relying on task-irrelevant signals is more pronounced than indicated by [18]. Nevertheless, as can be seen in the last row of Table A.1, even when using the looser measurement bound our techniques still record signal-awareness improvements, albeit masked by the checker leniency.

References

[1] M. Allamanis, E. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *FSE*, 2015.

[2] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," ser. ICLR, 2018.

[3] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *ICML*, 2016.

[4] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *ACL*, 2016.

[5] J. Li, Y. Wang, M. Lyu, and I. King, "Code completion with neural attention and pointer networks," *IJCAI*, 2018.

[6] GitHub Copolit, "Your AI pair programmer," 2021, https://copilot.github.com/.

[7] S. TEAM, "Accelerating our developer-first vision with deepcode," 2020, https://snyk.io/blog/accelerating-developer-first-vision-with-deepcode/.

[8] J. Bader, S. S. Kim, F. S. Luan, S. Chandra, and E. Meijer, "AI in software engineering at facebook," *IEEE Softw.*, vol. 38, no. 4, pp. 52–61, 2021. [Online]. Available: https://doi.org/10.1109/MS.2021.3061664

[9] R. Russell, L. Kim, L. Hamilton *et al.*, "Automated vulnerability detection in source code using deep representation learning," in *ICMLA*, 2018.

[10] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *NDSS*, 2018.

[11] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *NeurIPS*, 2019.

[12] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," 2018.

[13] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for predicting vulnerable software components," *IEEE Trans. Software Eng.*, vol. 47, no. 1, pp. 67–85, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2018.2881961

[14] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 19:1–19:29, 2019. [Online]. Available: https://doi.org/10.1145/3340544

[15] U. Yüksel and H. Sözer, "Automated classification of static code analysis alerts: A case study," in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ser. ICSM '13. USA: IEEE Computer Society, 2013, pp. 532–535.

[16] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin, "Aletheia: Improving the usability of static security analysis," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS'14*, 2014.

[17] U. Koc, P. Saadatpanah, J. S. Foster, and A. A. Porter, "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*, ser. MAPL 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 35–42.

[18] S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo, and A. Morari, "Probing model signal-awareness via prediction-preserving input minimization," ser. FSE, 2021.

[19] P. Bielik and M. T. Vechev, "Adversarial robustness for code," in *ICML*, 2020.

[20] M. R. I. Rabin, V. J. Hellendoorn, and M. A. Alipour, "Understanding neural code intelligence through program simplification," in *FSE*, 2021.

[21] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck, "Dos and don'ts of machine learning in computer security," *arXiv preprint arXiv:2010.09470*, 2020.

[22] M. Jimenez, R. Rwemalika, M. Papadakis, F. Sarro, Y. Le Traon, and M. Harman, "The importance of accounting for real-world labelling when predicting software vulnerabilities," in *FSE*, 2019.

[23] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *ACM SPLASH Onward!*, 2019.

[24] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, 2021.

[25] 0xabad1dea, "Risk assessment of github copilot," 2021. [Online]. Available: https://gist.github.com/0xabad1dea/be18e11beb2e12433d93475d72016902

[26] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *IEEE Symposium on Security and Privacy 2022*, 2022.

[27] A. A. Bangash, H. Sahar, A. Hindle, and K. Ali, "On the time-based conclusion stability of cross-project defect prediction models," *Empir. Softw. Eng.*, vol. 25, no. 6, pp. 5047–5083, 2020. [Online]. Available: https://doi.org/10.1007/s10664-020-09878-9

[28] H. Zhang, H. Chen, C. Xiao *et al.*, "Towards stable and efficient training of verifiably robust neural networks," in *ICLR*, 2019.

[29] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *ICLR*, 2018.

[30] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *ICLR*, 2015.

[31] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *ICML*, 2009.

[32] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Software Engineering*, vol. 28, no. 2, 2002.

[33] M. Pradel and K. Sen, "A learning approach to name-based bug detection," ser. OOPSLA, 2018.

[34] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Efficient javascript mutation testing," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013, pp. 74–83.

[35] D. B. Brown, M. Vaughn, B. Liblit, and T. Reps, "The care and feeding of wild-caught mutants," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 511–522. [Online]. Available: https://doi.org/10.1145/3106237.3106280

[36] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "Learning how to mutate source code from bug-fixes," in *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*. IEEE, 2019, pp. 301–312. [Online]. Available: https://doi.org/10.1109/ICSME.2019.00046

[37] B. Dolan-Gavitt, P. Hulin, E. Kirda *et al.*, "Lava: Large-scale automated vulnerability addition," in *IEEE S&P*, 2016, pp. 110–121.

[38] S. Roy, A. Pandey, B. Dolan-Gavitt, and Y. Hu, "Bug synthesis: Challenging bug-finding tools with deep faults," in *FSE*, 2018.

[39] J. Patra and M. Pradel, "Semantic bug seeding: A learning-based approach for creating realistic bugs," ser. FSE, 2021.

[40] "Clang Static Analyzer," https://clang-analyzer.llvm.org.

[41] "Pin - A Dynamic Binary Instrumentation Tool," https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[42] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[43] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy, SP'14*, 2014.

[44] C. Sestili, W. Snavely, and N. VanHoudnos, "Towards security defect prediction with AI," *CoRR*, vol. abs/1808.09897, 2018. [Online]. Available: http://arxiv.org/abs/1808.09897

[45] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[46] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[47] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[48] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *arXiv preprint arXiv:1506.01497*, 2015.

[49] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Thirtieth AAAI conference on artificial intelligence*, 2016.

[50] H. Sak, A. W. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," 2014.

[51] P. Liu, X. Qiu, and X. Huang, "Recurrent neural network for text classification with multi-task learning," *arXiv preprint arXiv:1605.05101*, 2016.

[52] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[53] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[54] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," *arXiv preprint arXiv:1802.09691*, 2018.

[55] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International Conference on Machine Learning*. PMLR, 2017, pp. 1263–1272.

[56] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence networks," in *ICLR*, 2016.

[57] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[58] V.-A. Nguyen, V. Nguyen, T. Le, Q. H. Tran, D. Phung *et al.*, "Regvd: Revisiting graph neural networks for vulnerability detection," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2022, pp. 178–182.

[59] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations (ICLR)*, 2020.

[60] "Juliet C/C++ 1.3 - NIST Software Assurance Reference Dataset," https://samate.nist.gov/SARD/test-suites/112.

[61] NIST, "Juliet test suite for c/c++ version 1.3," 2017, https://samate.nist.gov/SRD/testsuite.php.

[62] Facebook, "Infer Static Analyzer," 2015, https://fbinfer.com/.

[63] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 41–48.

[64] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c," in *International conference on software engineering and formal methods*. Springer, 2012, pp. 233–247.

[65] Lizard, "A simple code complexity analyser," 2012. [Online]. Available: https://github.com/terryyin/lizard

[66] Y. Zheng, S. Pujar, B. Lewis, L. Buratti *et al.*, "D2a: A dataset built for ai-based vulnerability detection methods using differential analysis," ser. ICSE-SEIP, 2021.

[67] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018.

[68] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity:learning graph transformations to detect and fix bugs in programs," in *ICLR*, 2020.

[69] S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo, and A. Morari, "Learning to map source code to software vulnerability using code-as-a-graph," *CoRR*, vol. abs/2006.08614, 2020. [Online]. Available: https://arxiv.org/abs/2006.08614

[70] N. Chawla, K. Bowyer, L. Hall, and W. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, 2002.

[71] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[72] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[73] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[74] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *International Conference on Machine Learning*. PMLR, 2020, pp. 5110–5121.

[75] G. Misherghi and Z. Su, "Hdd: Hierarchical delta debugging," in *Proceedings of the 28th International Conference on Software Engineering, ICSE'06*, 2006.

[76] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 361–371.

[77] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 335–346.

[78] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. ACM, 2016, pp. 85–96.

[79] M. Alazab, S. Venkatraman, P. Watters, and M. Alazab, "Zero-day malware detection based on supervised learning algorithms of api call signatures," in *Proceedings of the Ninth Australasian Data Mining Conference-Volume 121*. Australian Computer Society, Inc., 2011, pp. 171–182.

[80] R. Malhotra, "Comparative analysis of statistical and machine learning methods for predicting faulty modules," *Applied Soft Computing*, vol. 21, pp. 286–297, 2014.

[81] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2010.

[82] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.

[83] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 914–919.

[84] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 837–847.

[85] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: bug detection with n-gram language models," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 708–719.

[86] S. Nessa, M. Abedin, W. E. Wong, L. Khan, and Y. Qi, "Software fault localization using n-gram analysis," in *International Conference on Wireless Algorithms, Systems, and Applications*. Springer, 2008, pp. 548–559.

[87] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[88] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 757–762.

[89] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–30, 2019.

[90] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations*, 2019.

[91] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck, "Dos and don'ts of machine learning in computer security," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3971–3988.

[92] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of Big Data*, 2019.

[93] T. Brown, B. Mann, N. Ryder *et al.*, "Language models are few-shot learners," in *NeurIPS*, 2020.

[94] L. Yu, W. Zhang, J. Wang, and Y. Yu, "Seqgan: Sequence generative adversarial nets with policy gradient," ser. AAAI, 2017.

[95] C. Laidlaw, S. Singla, and S. Feizi, "Perceptual adversarial robustness: Defense against unseen threat models," in *ICLR*, 2020.

[96] G. Hacohen and D. Weinshall, "On the power of curriculum learning in training deep networks," in *ICML*, 2019.

[97] A. Graves, M. G. Bellemare, J. Menick, R. Munos, and K. Kavukcuoglu, "Automated curriculum learning for neural networks," in *ICML*, 2017.

[98] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *ACL*, 2020.

[99] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," in *CVPR*, 2016.

[100] R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations via gradient-based localization," ser. ICCV, 2017.

[101] A. Shrikumar, P. Greenside, and A. Kundaje, "Learning important features through propagating activation differences," in *ICML*, 2017.

[102] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep nets," ser. ICML, 2017.

[103] M. T. Ribeiro, S. Singh, and C. Guestrin, ""Why should I trust you?" Explaining the predictions of any classifier," in *KDD*, 2016.

[104] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, "Lemna: Explaining deep learning based security applications," in *CCS*, 2018.

[105] S. Lundberg and S. Lee, "A unified approach to interpreting model predictions," in *NIPS*, 2017.

[106] J. Adebayo, J. Gilmer, M. Muelly, I. Goodfellow, M. Hardt, and B. Kim, "Sanity checks for saliency maps," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 2018, pp. 9525–9536.

[107] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations, ICLR'18*, 2018.

[108] T. Xie and J. C. Grossman, "Crystal graph convolutional neural networks for an accurate and interpretable prediction of material properties," *Physical review letters*, vol. 120, no. 14, 2018.

[109] R. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "Gnnexplainer: Generating explanations for graph neural networks," 2019.

[110] G. Yan, S. Chen, Y. Bail, and X. Li, "Can deep learning models learn the vulnerable patterns for vulnerability detection?" in *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2022, pp. 904–913.

[111] Y. Ding, S. Suneja, Y. Zheng, J. Laredo, A. Morari, G. Kaiser, and B. Ray, "Velvet: a novel ensemble learning approach to automatically locate vulnerable statements," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 959–970.